Alexandre Guidet

Programmation pour smartphones et tablettes

Principes et applications

- Android, iOS, Windows 10
- Langages Java, Swift, C++, C#, Javascript





Références sciences

Programmation pour smartphones et tablettes

Principes et applications
Pour Android, IOS, Windows 10,
Langages Java, Swift, C++, C#, Javascript

Alexandre Guidet



Avant-propos

Les téléphones dits « intelligents » (*smartphones* en anglais) ont envahi notre vie quotidienne. En plus de leur fonction principale (qui reste téléphoner normalement) ils permettent d'écouter de la musique, de naviguer sur Internet, de jouer, de réserver un billet de train, d'effectuer un achat, de prendre des photos, de lire un livre, de regarder un film ou la télévision, ainsi qu'un nombre de plus en plus croissant d'autres usages. Pourquoi ? Tout simplement parce que nos téléphones sont en fait des ordinateurs, certes moins puissants, certes plus petits, mais qui ont l'énorme avantage d'être en permanence avec nous. Comme nos ordinateurs, ils peuvent donc être programmés, c'est-à-dire effectuer des tâches qui n'ont pas été prévues au départ par le constructeur.

Les fabricants de téléphones ne s'y sont pas trompés : ce qui est mis en avant n'est pas la fonction première (téléphoner) mais toutes les applications disponibles directement sur l'appareil.

Chez les informaticiens, les offres d'emploi de développeur mobile ont explosé ces dernières années. De nombreux éditeurs logiciels spécialisés dans le mobile sont apparus. Et bien entendu, les programmeurs amateurs se sont emparés de ce nouvel outil !

En tant qu'enseignant en informatique (notamment en programmation) dans un IUT[1], je me suis tout naturellement penché sur ces nouveaux ordinateurs et la manière de les programmer. Programmer pour un téléphone reste de la programmation, mais il y a un grand nombre de spécificités qui font qu'il n'est pas toujours aisé de transposer les connaissances que l'on a en programmation « classique » à la programmation « mobile ». Au sein du DUT[2] informatique, par exemple, un module entier consacré à la programmation mobile est apparu récemment. De nombreuses licences professionnelles dédiées à la programmation mobile ont été créées, ainsi que des masters universitaires et des parcours en écoles d'ingénieurs.

---

[1] Institut Universitaire de Technologie.
[2] Diplôme Universitaire de Technologie (niveau Bac+2).

Le but de ce livre est de décrire la programmation mobile, au travers d'exemples pratiques, en étudiant les différentes parties constituant une application mobile. De nombreux ouvrages existent mais sont en fait consacrés à un seul outil, un seul langage, une seule plateforme. Or il existe plusieurs systèmes mobiles, de nombreux langages de programmation, et de plus les outils évoluent très rapidement. J'ai donc voulu centrer ce livre sur la programmation mobile en général, et non sur un outil particulier.

Vous trouverez tout d'abord dans cet ouvrage une description rapide des différentes plateformes mobiles existantes à ce jour, un résumé des différents modes de développements utilisables ainsi qu'une description des outils et langages qui seront utilisés tout au long du livre.

Le livre abordera ensuite la programmation des interfaces utilisateur, sur toutes les plateformes et avec tous les outils décrits, puis le stockage des données, les capteurs présents sur nos téléphones (GPS, appareil photo…), la gestion du temps, les notifications utilisateur et les services connectés (*webservices*).

Une partie importante sera consacrée à la publication (dans les magasins d'application, les fameux *stores*) des applications, toujours bien entendu sur les différentes plateformes et à l'aide des différents outils étudiés.

Pour terminer, je vous présenterai un certain nombre d'ateliers pratiques : des applications réalisées entièrement, de la conception à la publication.

Le but avoué de ce livre est de vous faire découvrir la programmation mobile, de vous aider à choisir l'outil le plus adapté pour vous, suivant vos goûts, vos connaissances, le téléphone que vous voulez cibler.

Tout au long de l'ouvrage, j'utiliserai quelques conventions de notation pour faciliter la lecture.

Un mot dans une autre langue que le français (anglais la plupart du temps) sera écrit comme suit : *this is an English sentence.*

Un mot issu d'un mot-clé d'un langage informatique, ou d'un extrait de code, sera écrit avec une police différente, comme suit : `word`.

Un bloc complet de code sera indiqué comme ci-dessous :

```
var bloc = function(){
    alert("coucou :") ; // fait coucou !
}
```

JavaScript

Dans ce bloc, les mots-clés du langage choisi seront indiqués comme suit (**keyword**) et les commentaires avec une autre norme (***commentaire***). Une icône indiquera le langage utilisé, le livre en comportant plusieurs.

Le tableau ci-dessous donne les logos des langages de programmation employés :

Java	Swift	C++		JavaScript	php
Java	Swift	C#	C++	JavaScript	PHP

Tableau 1 : logos des langages utilisés

Des langages de description de fichier seront également utilisés :

HTML	CSS	XAML	XML	{JSON}
HTML	CSS	XAML	XML	JSON

Tableau 2 : logos des langages de description utilisés

Je tenais à terminer cette introduction en remerciant Sabine pour sa relecture patiente et pertinente, mes collègues de l'IUT pour les échanges réguliers et les discussions techniques toujours intéressantes, mes étudiants qui, par leurs questions en travaux pratiques, m'ont aidé à construire cet ouvrage, ainsi que vous, cher lecteur, pour avoir acheté ce livre, lu cette introduction jusqu'au bout et vous apprêter à entrer dans le vif du sujet, dès le chapitre suivant…

Table des matières

I.Les systèmes mobiles

Chaque téléphone intelligent, comme tout ordinateur, possède un **système d’exploitation** : c’est le logiciel qui est exécuté dès le démarrage de la machine et qui s’occupe de la gestion complète de celle-ci, notamment du lancement des autres logiciels (applications).

De même qu’il existe plusieurs systèmes d’exploitation pour ordinateurs personnels (Windows, macOS, Linux étant les plus connus), il existe plusieurs systèmes d’exploitation pour téléphones, la plupart étant également utilisés dans les tablettes tactiles, voire sur certains ordinateurs.

1. Android

Android est un système d’exploitation spécialisé au départ pour les téléphones mobiles. Datant de 2007 et développé par la société Google (à partir d’un projet développé par une start-up rachetée par Google), il est basé sur le noyau[1] Linux.

Android a fait le choix de la plateforme Java pour le développement, mais sans utiliser la machine virtuelle Java standard (propriété de la société Oracle). Jusqu’à la version 4.4 la machine virtuelle s’appelle *Dalvik*, puis ART (*Android RunTime*) à partir de la version 5.0. La bibliothèque standard d’Android ressemble beaucoup à J2SE (*Java 2 Standard Edition*) utilisé pour le développement « classique » sur ordinateurs, ce qui permet aux développeurs Java d’être familiarisés rapidement avec l’environnement.

D’abord destiné aux téléphones, le système Android est à présent disponible sur des tablettes, des ordinateurs de type PC, des montres, des téléviseurs, des box Internet…

C’est le système d’exploitation le plus utilisé dans le monde (en nombre d’appareils), c’est pourquoi il tiendra une place de choix dans cet ouvrage.

Les versions majeures d’Android se succèdent au rythme d’environ une par an.

---

[1] Le noyau d’un système d’exploitation est la partie chargée de la gestion du matériel, de la mémoire, de l’exécution des tâches.

2. iOS

iOS est le système d'exploitation, créé par Apple, qui équipe les appareils mobiles de la marque à la pomme : iPhone, iPad, iPod. C'est un système dérivé de macOS (le système d'exploitation des ordinateurs Apple), créé en 2007. La version actuelle est la 11, le système évoluant au rythme d'une version par an environ (les mises à jour de sécurité sont beaucoup plus fréquentes, évidemment). Il est obligatoire d'utiliser un ordinateur Apple sous macOS pour le développement d'applications pour iOS.

3. Windows 10

Windows est un système d'exploitation initialement destiné aux ordinateurs personnels de type PC, mais différentes branches de ce système ont été réalisées à destination des appareils mobiles (PDA puis téléphones).

Depuis la version 10 de Windows, son éditeur Microsoft a unifié ses différentes branches : c'est la plateforme Windows universelle (UWP : *Universal Windows Platform*) qui permet d'exécuter avec le même code, des applications sur PC, téléphone mobile, console de jeu Xbox, etc.

La présence de Windows sur des téléphones, bien qu'ancienne, n'a jamais été à la hauteur des deux autres plateformes (Android et iOS) et est en chute libre ces dernières années. Néanmoins, le côté universel de la plateforme la rend intéressante : développer pour téléphone n'est ici plus spécifique ! C'est pourquoi il tiendra une place dans cet ouvrage malgré des parts de marché faibles sur téléphone (mais meilleures sur tablettes, et très bonnes sur ordinateurs).

Il est obligatoire d'utiliser un ordinateur sous Windows 10 pour pouvoir développer pour un appareil Windows 10.

4. Ancêtres et exotiques

Malgré ce que pense souvent le grand public, les téléphones intelligents ne sont pas apparus avec l'iPhone ! Avant même les premiers téléphones programmables, il existait des appareils qui, mis à part la fonction de téléphonie, en étaient proches : les PDA (*personal digital assistant*).

Un des premiers systèmes d'exploitation pour appareils mobiles était *Palm OS*. Créé en 1995, ce système d'exploitation embarqué était utilisé



Image 1 : PDA sous PalmOS

dans des PDA, des smartphones, des montres, des GPS de voiture, des lecteurs de code-barres, des consoles de jeux. Il utilisait un écran tactile utilisable au doigt ou à l'aide d'un stylet. La programmation se faisait en général en C ou C++, mais il existait une machine virtuelle Java pour *Palm OS* (donc la possibilité de faire des applications en Java), des compilateurs Pascal et Basic et quelques autres langages plus exotiques. *Palm OS* est ensuite devenu *Palm WebOS*, puis *HP WebOS* et les derniers appareils sous ce système sont sortis en 2011.

Microsoft, leader dans les systèmes d'exploitation pour ordinateurs, a sorti en 1996 un système d'exploitation spécifique pour appareils mobiles appelé *Windows CE*. Utilisé dans des PDA, des téléphones, des DAB[1], des terminaux de paiement, des lecteurs de code-barres, des consoles de jeu, des GPS de voiture, le système a été très utilisé dans les premières générations d'appareils mobiles. Le système a ensuite évolué en *Windows mobile* mais a été abandonné en 2012 pour être remplacé par *Windows phone* (basé lui sur un autre noyau, donc totalement différent malgré le nom proche), lui-même remplacé par *Windows 10 universel*.



Image 2 : PDA sous Windows CE

La fondation Mozilla, éditrice du navigateur *Firefox*, a conçu un système d'exploitation pour appareils mobiles appelé *Firefox OS*. Il est basé, comme Android, sur le noyau Linux et se comporte en gros comme un navigateur : les applications doivent donc être écrites en HTML5. La première version date de 2013. Très peu de smartphones ont été vendus avec *Firefox OS*, mais le projet est toujours actif.

---

[1] Distributeur Automatique de Billets.

Il existe d'autres systèmes pour mobile, en général dérivés d'autres systèmes d'exploitation ou de navigateurs : *Ubuntu Touch*, *Sailfish OS*, *Tizen*, *Chromium OS*, *Google Chrome OS*…

II. Rappels de programmation objet

Le paradigme de programmation objet étant particulièrement employé dans la programmation sur terminaux mobiles (téléphones, tablettes…), comme dans d'autres domaines, quelques rappels me semblent importants. Si bien entendu vous êtes à l'aise avec ces notions, n'hésitez pas à passer directement au chapitre suivant !

1. Notion d'objet, de classe

Un **objet** est une représentation d'une notion du monde réel. C'est une variable, au sens programmation du terme, occupant une certaine zone de la mémoire centrale de la machine. Le paradigme objet revient, en gros, à considérer un programme comme un ensemble d'objets qui évoluent, communiquent et collaborent pour effectuer le travail voulu.

Un objet possède un certain **état** qui, au long du déroulement du programme, change. L'ensemble des états de tous les objets du programme représente l'état global du programme. Un programme peut alors être vu comme une machine à états, un automate.

Un objet possède également un certain **comportement**, c'est-à-dire un ensemble d'opérations qui permettent de modifier son état ou de collaborer avec l'objet.

Une **classe** est en quelque sorte le regroupement des objets de même nature (même type d'état, même comportement). C'est un **type** qui caractérise l'objet.

Une classe possède donc :

- Des **attributs** qui représentent son état : ce sont des variables
- Des **opérations** qui représentent son comportement ; on peut les regrouper en différents types :
    - Les **constructeurs** définissent l'état initial de l'objet
    - Les **mutateurs** modifient l'état de l'objet
    - Les **accesseurs** permettent de connaitre l'état d'un objet
    - Les **destructeurs** mettent fin à la vie de l'objet

Chaque membre d'une classe possède un certain niveau d'accès. Les niveaux courants sont les suivants :

- **Public** : tout objet peut accéder au membre
- **Protégé** : seuls les objets du type ou d'un type hérité peuvent accéder au membre
- **Privé** : seuls les objets de la même classe peuvent accéder au membre
- **Paquet** : seuls les objets du même paquet peuvent accéder au membre

Les accès courants sont :

- Privé pour les attributs et les opérations d'implémentation (détails du code)
- Public pour les opérations de l'interface
- Protégé pour les opérations d'implémentation abstraites ou les propriétés internes.

a) Représentation UML

Le langage UML est un langage de représentation graphique. Il est particulièrement adapté au paradigme objet. Une classe en UML se représente par un rectangle comportant trois compartiments :

- Le nom de la classe (avec un éventuel stéréotype)
- Les attributs de la classe
- Les opérations de la classe

Un attribut possède un certain type (entier, caractère, etc.), peut avoir une valeur initiale ainsi que certaines caractéristiques (constant, unique, etc.).

Horaire

-nbHeures : int
-nbMinutes : int
-nbSecondes : int

+Horaire()
+Horaire(h:int, m:int, s:int)
+Avancer(nbs : int)
+Convertir() : String
-Normaliser() : void

Diagramme 1 : une classe en UML

Une opération, parfois appelée méthode ou fonction, possède un certain type de retour (`void` si aucun), des paramètres et certaines caractéristiques.

La classe ci-contre (`Horaire`) représente un horaire dans la journée. Elle possède en attributs trois entiers qui permettent de stocker son état (nombre d'heures, de minutes, de secondes). Son

comportement comprend deux constructeurs (l'un sans paramètre et l'autre avec trois paramètres entiers), un accesseur (pour obtenir l'état sous la forme de texte) et deux mutateurs (un pour faire avancer l'horaire, et l'autre pour le normaliser) dont l'un, privé, est réservé à l'usage interne de la classe.

b) Une classe en Java

Le langage Java, utilisé notamment pour la programmation native Android (voir page 49), suit le paradigme objet et peut donc représenter une classe.

L'extrait de code suivant correspond au Diagramme 1 :

```
class Horaire {
     private int nbHeures;
     private int nbMinutes;
     private int nbSecondes;

     public Horaire()
     {
        nbHeures = nbMinutes = nbSecondes = 0;
     }
     public Horaire(int h, int m, int s)
     {
        nbHeures = h;
        nbMinutes = m;
        nbSecondes = s;
        normaliser();
     }
     public void avancer(int nbs)
     {
        nbSecondes += nbs;
        normaliser();
     }
     public String convertir()      {
        return String.valueOf(nbHeures)+
          ":"+String.valueOf(nbMinutes)+
          ":"+String.valueOf(nbSecondes)
     }
     private void normaliser(){
        nbMinutes += nbSecondes/60;
        nbSecondes %= 60;
        nbHeures += nbMinutes/60;
        nbMinutes %=60;
        nbHeures %= 24;
     }
}
```

c) Une classe en Swift

Swift est le langage natif pour développer sous iOS. En dehors de l'univers Apple, c'est un langage très peu utilisé et il n'est donc pas utile d'apprendre ce langage si vous ne développez pas pour les appareils à la pomme.

L'extrait de code suivant correspond au Diagramme 1 :

Swift

```
public class Horaire {
    private var nbHeures : Int
    private var nbMinutes : Int
    private var nbSecondes : Int

    public init()     {
        nbHeures = 0
        nbMinutes = 0
        nbSecondes = 0
    }

    public init(h:Int, m:Int, s:Int)
    {
        nbHeures = h
        nbMinutes = m
        nbSecondes = s
    }

    public func avancer(nbs:Int)
    {
        nbSecondes += nbs
        normaliser()
    }

    public func convertir() -> String
    {
        return String(nbHeures) + ":"  + String(nbMinutes) +
     ":" + String(nbSecondes)
    }

    private func normaliser()    {
        nbMinutes += nbSecondes/60
        nbSecondes %= 60
        nbHeures += nbMinutes/60
        nbMinutes %= 60
        nbHeures %= 24
    }
}
```

d) Une classe en C#

Le langage C# est notamment employé pour la plateforme .NET qui est utilisée pour le développement Windows mais aussi pour les plateformes mobiles Android et iOS à travers l'outil Xamarin (voir page 62). C# est assez proche syntaxiquement de Java et C++. La classe vue au a) s'écrira en C# de la manière suivante :

```
class Horaire
{
     private int nbHeures;
     private int nbMinutes;
     private int nbSecondes;

     public Horaire()
     {
        nbHeures = nbMinutes = nbSecondes = 0;
     }
     public Horaire(int h, int m, int s)
     {
        nbHeures = h;
        nbMinutes = m;
        nbSecondes = s;
        Normaliser();
     }
     public void Avancer(int nbs)
     {
        nbSecondes += nbs;
        Normaliser();
     }
     public String Convertir()
     {
        return nbHeures.ToString()
           + ":" + nbMinutes.ToString()
           + ":" + nbSecondes.ToString();
     }
     private void Normaliser()
     {
        nbMinutes += nbSecondes/60;
        nbSecondes %= 60;
        nbHeures += nbMinutes/60;
        nbMinutes %=60;
        nbHeures %= 24;
     }
}
```

e) Une classe en C++

C++ est un langage très utilisé qui, entre autres, implémente le paradigme objet. Il peut servir à développer des applications mobiles à travers notamment l'outil Qt (voir page 71).

C++ utilise deux fichiers pour une classe : un fichier de déclaration de la classe, ne contenant que très peu de code mais qui sert à déclarer le contenu de celle-ci (extension fréquente : `h`), et un fichier de définition de la classe (extension fréquente : `cpp`) qui contient le code des opérations de celle-ci.

La classe représentée en UML au a) se déclare en C++ de la manière suivante :

```
class horaire
{
 private:
     int nb_heures;
     int nb_minutes;
     int nb_secondes;
    void normaliser();
 public:
     horaire();
     horaire(int h, int m, int s);
     std::string convertir() const;
     void avancer(int nbs);
};
```

Elle se définit comme ci-dessous :

```
horaire::horaire():nb_heures(0),nb_minutes(0),nb_secondes(0)
{}

horaire::horaire(int h, int m, int s):
     nb_heures(h),nb_minutes(m),nb_secondes(s)
{
     normaliser();
}

void horaire::avancer(int nbs)
{
     nb_secondes += nbr;
     normaliser();
}
```

```
void horaire::normaliser() {
     nb_minutes += nb_secondes/60;
     nb_secondes %= 60;
     nb_heures += nb_minutes/60;
     nb_minutes %= 60;
     nb_heures %= 24;
}

std::string horaire::convertir() const{
     std::ostringstream flux;
     flux << nb_heures << ':' << nb_minutes << ':' <<
     nb_secondes;
     return flux.str();
}
```

f) Une classe en Javascript

Javascript, utilisé notamment pour le développement côté client des applications web mais aussi, grâce à l'outil Cordova (voir page 70), pour les applications mobiles, est un langage assez différent de Swift, C++, C# et même Java (malgré son nom).

En fait, en Javascript, les membres peuvent être ajoutés à l'objet à n'importe quel moment. Ce qui ressemble le plus à une classe « normale », c'est d'ajouter les différents attributs à l'objet `this` durant la construction. Il n'y a pas d'encapsulation, donc pas d'accès privé ou protégé.

Les variables n'étant pas typées en Javascript, la syntaxe est assez différente des langages plus classiques comme Java ou C++. Le mot-clé `this` est ainsi obligatoire pour accéder à un membre (attribut ou opération)

La classe vue plus haut se déclare comme suit en Javascript :

```
class Horaire {
     constructor(h=0, m=0, s=0)
     {
       this.nbHeures = h;
       this.nbMinutes = m;
       this.nbSecondes = s;
     }
     avancer(nbs)
     {
       this.nbSecondes += nbs;
       this.normaliser();
     }
     convertir()
```

```
    {
      return this.nbHeures.toString() + ":" +
    this.nbMinutes.toString() + ":"+this.nbSecondes;
    }
    normaliser()
    {
      this.nbMinutes += Math.floor(this.nbSecondes/60);
      this.nbSecondes %= 60;
      this.nbHeures += Math.floor(this.nbMinutes/60);
      this.nbMinutes %= 60;
      this.nbHeures %= 24;
    }
}
```

2. Associations entre classes

Les objets d'un programme vont éventuellement collaborer les uns avec les autres. Pour collaborer, un objet envoie un message à un autre objet. « Envoyer un message » est un autre terme pour dire « appeler une opération ». Il faut donc que l'objet puisse avoir une référence vers l'objet auquel il doit envoyer un message. Cette référence peut être transitoire, dans une opération dont un paramètre est l'objet ciblé, ou persistante, si l'objet est associé avec l'autre.

Une classe est dite associée avec une autre si les instances de la première classe peuvent envoyer des messages aux instances de la deuxième.

a) Représentation UML

Dans un diagramme UML, les deux classes seront reliées par une association. Chaque terminaison d'une association est donc une classe. Une terminaison possède des caractéristiques :

- Un nom, qui caractérise le rôle de la classe dans l'association
- Une cardinalité, qui indique le nombre d'instances mises en jeu dans l'association

Une terminaison peut être vide, dans ce cas l'association est à un seul sens.

L'exemple ci-dessous montre l'association entre une classe `Personne` et une classe `Logement`. Du côté de la classe `Personne`, le rôle est `habitant`, et la cardinalité est « plusieurs ». Du côté de la classe `Logement`, le rôle est `habitation`, et la cardinalité est « un ou aucun ».



Diagramme 2 : association entre deux classes

En plus de ce type (association simple) il existe une notion proche : l'agrégation. C'est une association asymétrique dans laquelle la classe du côté losange est un contenant de l'autre :



Diagramme 3 : agrégation

Il existe un troisième type : la composition. C'est une agrégation dans laquelle le contenu est lié fortement au contenant, et vice-versa : la disparition de l'un entraîne la disparition de l'autre.

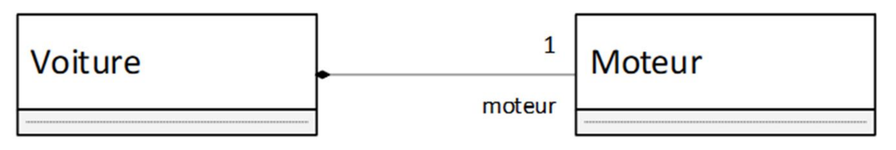


Diagramme 4 : composition

b) Associations en Java

En Java, associations, agrégations et compositions se représentent de la même manière. Une terminaison identifiée (avec un rôle et une cardinalité) va apporter la création d'un **attribut** dans la classe opposée. Cet attribut est simple, dans le cas où la cardinalité maximum est de 1, et une **collection** dans les autres cas (plusieurs éléments associés). Les collections possibles sont nombreuses, mais la plupart du temps on utilisera `ArrayList` (tableau), `LinkedList` (liste chaînée), `TreeSet` (arbre de recherche), ou `HashMap` (table de hachage). Comme tout attribut, son mode d'accès sera privé. La classe devra fournir des opérations pour la mise à jour de l'association.

L'association vue au a) se représentera en Java avec le code suivant :

```
class Personne{
    private Logement habitation=null;
}
class Logement{
    private ArrayList<Personne> habitants = new
    ArrayList<>();
}
```

c) Associations en Swift

Comme en Java, une association en Swift revient à un attribut simple (association avec un seul objet) ou une collection (association avec plusieurs objets). Les collections utilisables dans ce cas sont `Array` (tableau), `Set` (ensemble, type arbre binaire de recherche) ou `Dictionary` (tableau associatif).

L'association vue au a) se représentera en Swift avec le code suivant :

```
class Personne
{
    private var habitation : Logement = nil
}

class Logement
{
    private var habitants : [Personne]()
}
```

d) Associations en C#

C# étant très proche syntaxiquement de Java, la représentation des associations est tout naturellement quasiment identique. Les collections sont différentes : nous utiliserons en C# `List` (à la fois tableau et liste chaînée) et `Dictionnary` (tableau associatif). Le code C# est le suivant :

```
class Personne{
    private Logement habitation=null;
}

class Logement{
    private List<Personne> habitants = new List<Personne>();
}
```

e) Associations en C++

La représentation des associations est assez proche de celle vue pour Java, Swift et C#. Néanmoins, il y a des différences. Tout d'abord, en C++, l'attribut représentant l'association sera un pointeur (comme en Java, mais en Java ça ne se voit pas). Ensuite, dans le cas des compositions, il faudra gérer la libération mémoire (inutile en Java, Swift ou C#).

Les collections C++ les plus courantes sont `vector` (tableau), `list` (liste chaînée), `set` (ensemble, type arbre de recherche) et `map` (tableau associatif). D'autres sont bien entendu possibles, mais dans tous les cas il faut manipuler des pointeurs (les pointeurs « intelligents », nouveauté du C++ 11, ne sont pas vus ici pour simplifier, mais ils peuvent bien entendus être utilisés, notamment pour automatiser la libération mémoire).

La traduction en C++ du diagramme vu page 20 est la suivante :

```
class Personne
{
 private:
     Logement* habitation;
};

class Logement
{
 private:
     std::set<Personne*> habitants;
};
```

La composition est un cas particulier en C++ : celle-ci impose en effet un lien de cycle de vie, et donc une obligation de libération mémoire. Si nous prenons l'exemple vu plus haut (voiture et moteur), la composition implique que la destruction (la fin de vie) de l'objet `Voiture` doit entraîner la destruction de l'objet `Moteur` contenu. Nous trouverons donc obligatoirement, dans la classe `Voiture`, un **destructeur** :

```
~Voiture(){
     delete moteur;
}
```

Pour bien faire les choses, il faudrait également traiter le cas de la copie : en effet, en C++, la copie est automatique mais se contente de recopier les attributs. Dans le cas d'une composition, cela va poser un problème ! Mais

ce n'est pas le but de cet ouvrage, je vous invite à consulter (1) pour plus de détails.

f) Associations en Javascript

Bien que Javascript ne code pas exactement des attributs (en tout cas, pas de la même manière que Java), la représentation d'une association va être proche de ce que nous avons vu plus haut. Dans le cas d'une cardinalité multiple, nous utiliserons un tableau Javascript (`Array`). Hélas, Javascript n'est que très faiblement typé, et donc notre association n'est pas non plus typée…

Le code suivant indique comment représenter (partiellement) notre association :

```
class Personne
{
     constructor(){
       this.logement = null;
     }
}

class Logement
{
     constructor(){
       this.habitants = []; // tableau Javascript vide
     }
}
```

3. Héritage, polymorphisme

L'héritage est une notion centrale en programmation objet. Elle consiste à préciser, lors de la création d'une classe, la classe servant de base, ou d'ancêtre. Une classe récupère (hérite) toutes les fonctionnalités de son ancêtre, état comme comportement.

Si une classe B hérite d'une classe A, par exemple, tous les membres de A sont disponibles dans B. Un objet du type B peut donc être vu comme un objet du type A, puisque le type B est au moins équivalent au type A. On appelle ce principe le **polymorphisme**.

L'héritage est fréquemment utilisé pour effectuer une généralisation.

Les opérations d'une classe A peuvent être **redéfinies** dans la classe héritière B.

a) Représentation UML

L'héritage en UML est représenté par une flèche pleine de la classe héritière vers sa classe ancêtre. La flèche va donc du cas particulier au cas général.

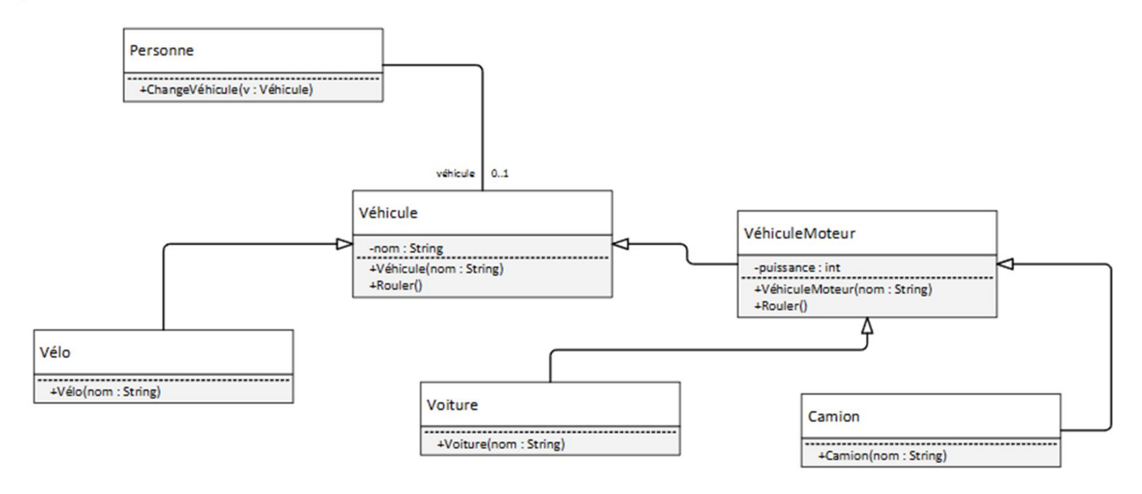


Diagramme 5 : représentation UML de l'héritage

Dans le diagramme précédent, on trouve une classe `Véhicule` (cas général). Une `Personne` peut conduire un `Véhicule` (voir l'association), ce véhicule pouvant être de différents types. L'association ici se fait avec un type général, et permet donc de traiter différents cas particuliers avec un seul cas. L'opération `Rouler`, présente dans la classe la plus générale, est donc utilisable avec tout objet hérité. Si la personne conduit un vélo, par exemple, elle pourra appeler l'opération `Rouler`. Cette opération a été redéfinie dans la classe `VéhiculeMoteur` car elle va être différente dans cette classe. Les classes `Voiture` et `Camion`, en revanche, ne redéfinissent pas cette opération : un objet de type `Voiture` utilisera donc la fonction `Rouler` de son ancêtre, la classe `VéhiculeMoteur`.

On remarque ici tout l'intérêt de la notion de **polymorphisme** : un objet de type `Véhicule`, associé avec `Personne`, peut être en fait un `Vélo`, une `Voiture`, un `Camion`. Si on appelle la fonction `Rouler` du véhicule associé avec la personne, c'est la fonction `Rouler` la plus précise (le cas particulier) qui sera appelée !

Cette notion est assez complexe et pourrait être décrite dans un chapitre complet ! Ce n'est pas le propos de cet ouvrage, si la notion n'est pas très claire pour vous, je vous invite à consulter d'autres ouvrages / sites web.

b) Héritage en Java

Une classe Java ne peut hériter que d'une seule classe (pas d'ancêtres multiples). Si aucune classe n'est précisée, la classe hérite de `Object` qui est une classe du JDK, ancêtre commun de toutes les classes Java.

L'héritage utilise le mot-clé `extends`. Dans une opération, on se sert du mot-clé `super` pour faire référence à l'ancêtre (la superclasse).

Le code ci-dessous montre l'implémentation Java d'une partie du Diagramme 5 :

```java
class Véhicule
{
 private String nom;
 public Véhicule(String nom)
 {
     this.nom = nom;
 }
 public void rouler()
 {
     System.out.println("Je roule !");
 }
}
```

```java
class VéhiculeMoteur extends Véhicule {
 private int puissance;
 public VéhiculeMoteur(String nom, int puissance)
 {
     super(nom);
     this.puissance = puissance;
 }
 @Override
 public void rouler()
 {
     super.rouler() ;
     System.out.println(" Avec un moteur");
 }
}
```

L'utilisation du polymorphisme est simple :

```
Véhicule v = new VéhiculeMoteur("Test",10);
v.rouler(); // affiche "Je roule ! Avec un moteur"
```

c) Héritage en Swift

Swift ne gère pas plus l'héritage multiple que Java.

Le mot-clé utilisé pour la classe ancêtre est là aussi `super` ; Swift utilise `self` à la place de `this` pour désigner l'instance utilisée. Une fonction redéfinie doit être précédée du mot-clé `override`.

```
class Véhicule
{
     private var nom : String
     public init(nom : String)
     {
        self.nom = nom
     }
     public func rouler()
     {
        print("Je roule !")
     }
}
```

```
class VéhiculeMoteur : Véhicule
{
     private var puissance : Int
     public init(nom: String, puissance:Int)
     {
        self.puissance = puissance
        super.init(nom : nom)
     }


     public override func rouler()
     {
        super.rouler()
        print(" Avec un moteur")
     }
}
```

```
var v:Véhicule = VéhiculeMoteur(nom :"Test",puissance :10)
v.rouler() // affiche Je roule ! Avec un moteur
```

d) Héritage en C#

De même que Java, C# n'autorise pas l'héritage multiple. Il existe également une classe `Object` qui est la superclasse par défaut. Les différences sont principalement syntaxiques, mais il y en a une importante : une fonction doit être indiquée `virtual` pour être redéfinissable. Quand elle est redéfinie, le mot-clé `override` doit être précisé.

Le code suivant montre l'implémentation C# d'une partie du Diagramme 5. On remarque que les différences avec Java sont assez légères :

```
class Véhicule
{
 private String nom;
 public Véhicule(String nom)
 {
     this.nom = nom;
 }
 virtual public void Rouler()
 {
     Console.Writeln("Je roule !");
 }
}
```

```
class VéhiculeMoteur : Véhicule {
 private int puissance;
 public VéhiculeMoteur(String nom, int puissance)
     : base(nom)  {
     this.puissance = puissance;
 }


 override public void Rouler()
 {
     base.Rouler() ;
     Console.Writeln(" Avec un moteur");
 }
}
Véhicule v = new VéhiculeMoteur("Test",10);
v.Rouler(); // affiche "Je roule ! Avec un moteur"
```

e) Héritage en C++

C++ autorise, lui, l'héritage multiple. Il permet beaucoup plus de choses, comme l'héritage privé ou protégé, mais nous n'avons pas la place de tout lister ici !

Attention : les variables C++ sont des valeurs (et non des références comme dans la plupart des langages objets), il faut donc impérativement utiliser des pointeurs ou des références pour permettre le polymorphisme.

Comme en C#, les fonctions doivent être `virtual` pour être redéfinissables. Lors de leur redéfinition, le mot-clé `override` est recommandé mais non obligatoire : apparu avec C++11, le code réalisé avec des versions antérieures n'est pas compatible.

Le code suivant montre l'implémentation C++ d'une partie du Diagramme 5 :

```
class Véhicule {
 private:
     string nom;
 public:
     Véhicule(string nom)    {
        this->nom = nom;
     }
     virtual ~Véhicule(){} // conseillé !
     virtual void Rouler()   {
        cout << "Je roule !" << endl;
     }
};
```

```
class VéhiculeMoteur : public Véhicule {
 private:
     int puissance;
 public:
     VéhiculeMoteur(string nom, int puissance) :
     Véhicule(nom) {
        this->puissance = puissance;
     }

     void Rouler() override  {
      Véhicule::Rouler();
      cout << " Avec un moteur" << endl;
     }
};
```

```
Véhicule *v = new VéhiculeMoteur("Test",10);
v->Rouler(); // affiche "Je roule ! Avec un moteur"
delete v ;
```

f) Héritage en Javascript

Javascript étant très peu typé, la notion de polymorphisme n'est pas du tout la même que dans les autres langages objet. Néanmoins l'héritage existe ! Mais il est très différent de celui de Java, C# ou C++ car il utilise initialement des **prototypes** plutôt que des classes. La dernière version (ECMAScript 6) introduit néanmoins une syntaxe assez proche de l'héritage Java.

Le code suivant montre l'implémentation Javascript (ECMAScript 6) d'une partie du Diagramme 5 :

```
class Véhicule {
     constructor(nom)
     {
        this.nom = nom;
     }
     rouler()
     {
        console.log("Je roule !");
     }
}
```

```
class VéhiculeMoteur extends Véhicule {
     constructor(nom, puissance)
     {
        super(nom);
        this.puissance = puissance;
     }
     rouler()
     {
        super.rouler();
        console.log(" Avec un moteur");
     }
}
```

```
var v = new VéhiculeMoteur("Test",10);
v.rouler();
```

4. Abstraction et interfaces

Il arrive que certaines classes, dans une hiérarchie d'héritage, soient tellement générales qu'aucun objet de leur type exact n'a de sens.

Prenons un exemple : des formes géométriques planes (2 dimensions). Toutes les formes ont une surface, mais il est impossible de la déterminer de manière générale. Nous modéliserons donc ce problème par une classe générale, représentant une forme, et des classes héritières (cercle, rectangle, etc.), mais aucune instance de `Forme` n'a de sens puisqu'il est impossible de donner une formule générale à la surface. On parle de fonction **abstraite** pour la surface, et de classe abstraite pour `Forme`. Toute classe contenant au moins une opération abstraite est abstraite.

Les opérations et les propriétés (qui sont en fait des opérations) peuvent être abstraites, mais pas les attributs, ni les constructeurs et destructeur.

Il est impossible de créer une instance d'une classe abstraite : celle-ci ne sert que pour l'héritage. Une classe héritant d'une classe abstraite doit redéfinir toutes les opérations abstraites pour être « normale », sinon elle reste abstraite !

L'interface est une sorte de classe totalement abstraite : elle ne contient aucun attribut, aucune opération concrète (donc aucun code), et même aucun membre non public. L'interface est en fait la représentation de fonctionnalités (comportement) possibles pour un objet.

Une classe n'hérite pas d'une interface, elle **implémente** celle-ci.

a) Représentation UML

En UML, une opération ou une classe abstraite porte un nom en italique.

Le diagramme ci-contre montre deux classes, l'une abstraite et l'autre concrète.

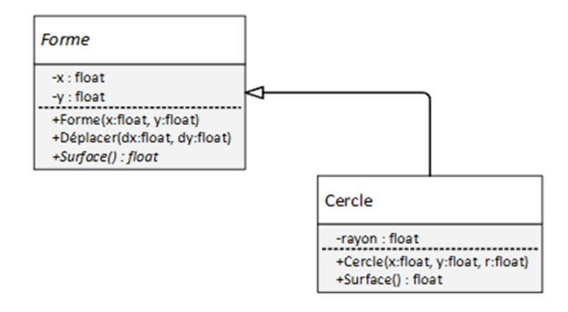


Diagramme 6 : abstraction en UML

La classe générale abstraite `Forme` contient :

- Des attributs représentant le point central de la forme

- Un constructeur qui initialise l'état de la forme (les attributs)
- Une opération concrète, le déplacement, qui consiste à modifier le point central de la forme.
- Une opération abstraite, la surface, impossible à généraliser

La classe héritée `Cercle`, concrète, contient :

- Un attribut supplémentaire, le rayon
- Un constructeur pour initialiser l'état (rayon, ainsi que point central)
- La redéfinition de l'opération abstraite de son ancêtre : on peut en effet à présent calculer la surface !

L'interface, en UML, se représente comme une classe mais possède en outre le stéréotype « `Interface` ». Ses membres étant obligatoirement publics et abstraits, ni le « + » ni l'italique ne sont précisés. Le diagramme suivant montre une interface représentant la fonctionnalité de gestion de sauvegarde (sur disque) d'une forme :

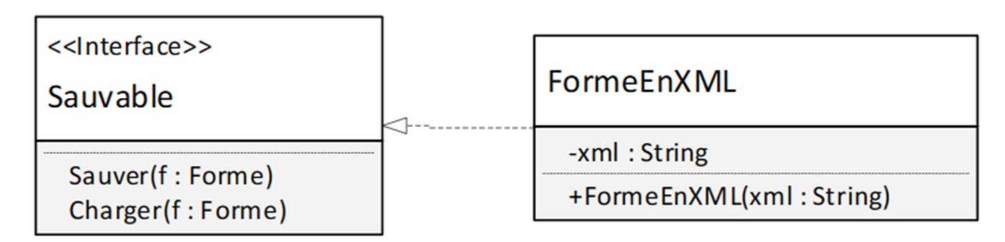


Diagramme 7 : interface en UML

La classe `FormeEnXML` est concrète. En UML, la flèche reliant une classe à une interface est différente de celle de l'héritage : elle est en pointillés. On ne précise pas dans `FormeEnXML` la redéfinition des opérations `Sauver` et `Charger` car celle-ci est obligatoire : elles sont obligatoirement présentes.

b) Abstraction en Java

Java utilise le mot-clé `abstract` pour déclarer une classe ou une opération abstraite. Une classe ne contenant aucune opération abstraite ne peut être déclarée `abstract`, et une classe contenant une opération abstraite au minium doit être déclarée `abstract`.

Le code ci-dessous est l'équivalent Java du Diagramme 6 :

```
abstract class Forme
{
     private float x;
     private float y;
     public Forme(float x, float y)
     {
        this.x = x;
        this.y = y;
     }
     public void déplacer(float dx, float dy)
     {
        x += dx;
        y += dy;
     }
     public abstract float surface();
}
```

```
class Cercle extends Forme
{
     private float rayon;
     public Cercle(float x, float y, float r)   {
        super(x,y);
        this.rayon = r;
     }

     public float surface()  {
        return Math.PI*rayon*rayon;
     }
}
```

Concernant les interfaces, Java utilise le mot-clé `interface` pour en déclarer une, et le mot-clé `implements` pour déclarer une classe implémentant l'interface. Une classe peut implémenter plusieurs interfaces.

Le code ci-dessous est l'équivalent Java du Diagramme 7 :

```
interface Sauvable
{
     void sauver(Forme f);
     void charger(Forme f);
}
```

```
class FormeEnXML implements Sauvable
{
     private String xml;
     public FormeEnXML(String xml){
        this.xml = xml;
     }
     public void sauver(Forme f)
     {
        // code de la sauvegarde XML
     }
     public void charger(Forme f){
        // code du chargement XML
     }
}
```

c) Abstraction en Swift

Swift ne possède pas réellement la notion de classe abstraite, ni d'interface, mais une notion proche appelée **protocole**. Un protocole en Swift indique des fonctionnalités que toute instance adoptant le protocole doit fournir. Il est donc très proche d'une interface. Un protocole peut être étendu en lui ajoutant des opérations concrètes, ce qui le fait ressembler à une classe abstraite. Des attributs ne peuvent cependant pas être ajoutés.

Le code ci-dessous est un équivalent Swift du Diagramme 6 :

```
protocol Forme{
    func surface()->Float
    var x : Float {get set}
    var y : Float {get set}
}
extension Forme{
    mutating func deplacer(dx:Float, dy:Float){
        x += dx
        y += dy
    }
}
```

Le protocole `Forme` est déclaré comme possédant une opération `surface` et deux propriétés `x` et `y`. Il est ensuite étendu pour lui ajouter une opération, `déplacer`, marquée comme `mutating` car elle modifie les attributs.

```
class Cercle : Forme{
    internal var y: Float
    internal var x: Float
    private var rayon: Float

    public init(x:Float, y:Float, rayon:Float)
     {
        self.x = x
        self.y = y
        self.rayon = rayon
    }
    public func surface() -> Float
     {
        return Float(M_PI)*rayon*rayon
    }
}
```

Swift

En Swift les attributs n'étant pas stockés dans le prototype, chaque classe l'implémentant devra les définir, alors que dans d'autres langages objets, une factorisation de code est possible.

Le protocole étant assez proche d'une interface, on peut pratiquement le considérer comme tel.

Le code ci-dessous est l'équivalent Swift du Diagramme 7 :

```
protocol Sauvable {
    func sauver(forme : Forme)
    func charger(forme : Forme)
}
```

Swift

```
class FormeEnXml : Sauvable {
    private var xml : String
    public init(xml:String)
     {
        self.xml = xml
    }

    public func charger(forme: Forme)     {
       // code du chargement XML
    }
    public func sauver(forme: Forme)      {
       // code de la sauvegarde XML
    }
}
```

Swift

d) Abstraction en C#

C# étant assez proche de Java, leurs syntaxes se ressemblent. Il n'y a pas de spécificités particulières liées à l'abstraction. Pour déclarer l'implémentation, on utilise « **:** », comme pour l'héritage. Les règles liées à l'héritage, à l'abstraction et à la réalisation d'interfaces sont les mêmes qu'en Java.

Le code ci-dessous est l'équivalent C# du Diagramme 6 :

```
abstract class Forme
{
     private float x;
     private float y;
     public Forme(float x, float y)
     {
        this.x = x;
        this.y = y;
     }
     public void Déplacer(float dx, float dy)
     {
        x += dx;
        y += dy;
     }
     public abstract float Surface();
}
```

```
class Cercle : Forme {
     private float rayon;
     public Cercle(float x, float y, float r):base(x,y)
     {
        this.rayon = r;
     }
     public float Surface()
     {
        return Math.PI*rayon*rayon;
     }
}
```

Concernant les interfaces, C# utilise également le mot-clé `interface`.

Le code ci-après est l'équivalent C# du Diagramme 7 :

```
interface Sauvable{
     void Sauver(Forme f);
     void Charger(Forme f);
}
```

```
class FormeEnXML : Sauvable {
     private String xml;
     public FormeEnXML(String xml){
        this.xml = xml;
     }
     public void Sauver(Forme f){
        // code de la sauvegarde XML
     }
     public void Charger(Forme f){
        // code du chargement XML
     }
}
```

e) Abstraction en C++

C++ n'utilise pas de mot-clé ni pour une classe abstraite, ni pour une opération abstraite. Pour représenter une opération abstraite, il faut qu'elle soit **virtuelle** (obligatoire pour être redéfinissable, voir e), mais au lieu d'écrire du code, on ajoute « =0 ». On parle alors d'opération « virtuelle pure ».

Le code ci-dessous est l'équivalent C++ du Diagramme 6 :

```
class Forme {
 private:
     float x;
     float y;
 public:
     Forme(float x, float y)  {
        this->x = x;
        this->y = y;
     }
     void Déplacer(float dx, float dy)
     {
        x += dx;
        y += dy;
     }
     virtual float Surface() const =0 ;
};
```

```
class Cercle : public Forme{
 private:
     float rayon;
 public:
     Cercle(float x, float y, float r):Forme(x,y){
        rayon = r;
     }
     float Surface() const {
        return Math.PI*rayon*rayon;
     }
};
```

Pour les interfaces, C++ est assez différent de Java ou C# : il ne possède pas de mot-clé `interface`. Pour C++, une interface est une classe, rien de plus. Et cela ne pose aucun problème, car C++ implémente l'héritage multiple !

Pour représenter une interface, on utilisera donc en C++ une classe ne contenant que des opérations virtuelles pures publiques. Il faudra également implémenter un destructeur virtuel vide, pour permettre aux classes implémentant l'interfaces d'avoir un éventuel destructeur (2).

Le code ci-dessous est l'équivalent C++ du Diagramme 7 :

```
class Sauvable {
public:
     virtual ~Sauvable(){}
     virtual void Sauver(Forme& f) =0;
     virtual void Charger(Forme& f) =0;
}
```

```
class FormeEnXML : public Sauvable {
 private:
     string xml;
 public:
     FormeEnXML(string xml){
        this->xml = xml;
     }
     void Sauver(Forme& f){    // code de la sauvegarde XML
     }
     void Charger(Forme& f){          // code du chargement XML
     }
}
```

f) Abstraction en Javascript

Javascript ne possède pas de mot-clé particulier pour déclarer une opération ou une classe comme abstraite. Il est néanmoins possible de le représenter simplement, en utilisant des exceptions.

Pour réaliser une opération abstraite, il suffit de réaliser une opération qui lève une exception. Ce n'est pas exactement une classe abstraite, mais cela peut en jouer le rôle. En particulier, si le constructeur lève une exception, il est impossible de créer une instance de la classe, ce qui revient bien à une classe abstraite (3).

Le code ci-dessous est l'équivalent Javascript du Diagramme 6 :

```
class Forme
{
     constructor(x,y)
     {
        if(this.constructor===Forme)
           throw new TypeError("Forme est abstraite");
        this.x = x;
        this.y = y;
     }
     déplacer(dx,dy)
     {
        this.x += dx;
        this.y += dy;
     }
     surface()
     {
        throw new Error("surface est abstraite dans Forme");
     }
}
```

```
class Cercle extends Forme
{
     constructor(x,y,rayon){
        super(x,y);
        this.rayon = rayon;
     }
     surface(){
        return Math.PI*rayon*rayon;
     }
}
```

Javascript ne possède pas la notion d'interface. Celle-ci est simulable (voir (3)) mais au prix d'un peu de gymnastique, par ce qu'on appelle le *duck typing*. Ce mot étrange (typage du canard) vient d'un test énoncé comme suit :

> Si ça ressemble à un canard, si ça nage comme un canard et si ça cancane comme un canard, c'est qu'il s'agit sans doute d'un canard.

Dit autrement, si un objet possède un certain nombre de caractéristiques propres à un type, c'est qu'il est probablement de ce type. Cela ressemble à la notion d'interface et peut donc être utilisé pour l'implémenter en Javascript au besoin.

III.Les modes de développement

1. Développement natif

Le développement appelé **natif** est celui préconisé par la plateforme : il utilise en général un environnement bien précis, défini par l'éditeur de la plateforme (Google pour Android, Apple pour iOS, Microsoft pour Windows). Il présente l'avantage d'être évidemment particulièrement adapté pour la plateforme ciblée, mais par conséquent il n'est absolument pas utilisable sur les autres plateformes. Si vous souhaitez développer pour plusieurs téléphones (par exemple, un Android et un iPhone) vous serez obligés d'utiliser plusieurs codes différents pour chaque plateforme : ceux-ci sont en effets natifs à chaque plateforme.

L'application peut utiliser toutes les fonctionnalités du téléphone, elle sera rapide à l'exécution, l'aspect visuel sera cohérent avec les autres applications, l'environnement de développement est adapté au déploiement direct dans le magasin applicatif.



Il est obligatoire de développer plusieurs versions en parallèle si l'on veut cibler plusieurs plateformes ; le langage de programmation est imposé, de même que l'environnement de développement; le kit de développement peut être complexe à utiliser.

Les environnements de développement natifs sont, à l'heure où ces lignes sont imprimées, les suivants :

- Pour Android : Android Studio 3.1. Disponible pour Windows, Linux, Mac OS. Utilise les langages Java et Kotlin. Gratuit.
- Pour Windows 10 mobile : Visual Studio 2017. Disponible pour Windows 10. Une version gratuite existe. Utilise principalement le langage C#.
- Pour iOS : XCode 9.3. Disponible pour Mac OS (10.8 minimum). Gratuit. Utilise le langage Objective C ou le langage Swift.

2. Multiplateforme web

La première approche pour les développeurs qui souhaitent utiliser le même code sous différentes plateformes est d'utiliser les technologies du web. En effet, tous les systèmes mobiles (ainsi que les systèmes classiques) possèdent un navigateur capable d'exécuter des applications web. Il faut bien entendu adapter l'affichage à un terminal mobile (écran plus petit, orientation portrait ou paysage) ainsi qu'aux interactions (clavier virtuel, écran tactile…) mais il est assez rapide de passer d'une application web « pour mobiles » à une application « mobile ».

Les langages utilisés sont ceux qui peuvent être interprétés par un navigateur : HTML 5 et CSS[1] 3 pour la présentation et Javascript / ECMAScript[2] pour le code métier.

Il est ainsi possible de faire une « application web » et de l'exécuter dans le navigateur d'un téléphone, mais ce n'est pas une « vraie » application : elle ne peut pas être installée via le magasin applicatif, ne peut pas utiliser certaines parties du téléphone, et présente une lenteur à l'exécution.

Certains outils, comme Cordova d'Apache, ont recours aux langages du web mais ajoutent un *framework* qui permet d'utiliser certains composants du téléphone inaccessibles au navigateur. De plus, l'outil réalise une vraie application qui peut être installée par le magasin applicatif. La lenteur d'exécution inhérente à l'interprétation Javascript est en revanche toujours présente.



Le même code peut être utilisé sur un grand nombre de plateformes ; l'application peut être issue d'un site web existant et adaptée très rapidement ; les compétences développeur acquises pour le web sont réutilisables directement.

L'exécution est un peu lente (par rapport à une application native) ; les fonctionnalités spécifiques à une plateforme ne sont pas utilisables ; l'aspect de l'interface peut être différent



[1] CSS : *Cascade StyleSheet* : fichier contenant les styles visuels d'une page HTML.
[2] JavaScript est le nom « grand public » du langage. Sa normalisation est ECMAScript.

des applications « classiques » ; les langages du web peuvent être difficiles à manipuler.

La solution du multiplateforme web est une bonne solution si l'application ne doit pas être très rapide, si le multiplateforme est obligatoire, si du code « web » existe déjà ou si le développeur possède une forte compétence dans ces langages. Si en revanche vous voulez écrire un jeu 3D, ou utiliser un capteur spécifique, ou si vous ne savez pas du tout coder en HTML5/CSS/Javascript, la solution peut être inadaptée et risque de faire perdre beaucoup de temps.

Ce type de développement est souvent privilégié par les éditeurs de logiciels simples dont l'application fait « doublon » avec le site web et doit être présente sur plusieurs plateformes (Android, Windows et iOS) en plus du web. Une très grande partie du code étant commun, le temps de développement et de maintenance est fortement réduit.

Parmi les outils existants, on peut citer :

- Apache Cordova (*open source*). C'est le *framework* que nous emploierons dans cet ouvrage pour le codage en HTML5 des applications mobiles. Il est notamment intégré à l'EDI Visual Studio 2017 (mais peut être utilisé sans, avec d'autres EDI ou en ligne de commande).
- Adobe Phonegap (basé sur Cordova)
- Appcelerator Titanium

3. Multiplateforme natif

L'idée est de combiner les avantages de la programmation native et de la programmation multiplateforme, en permettant, avec une grande base de code commune (voire l'intégralité du code), de développer sur plusieurs plateformes différentes. L'outil traduit dans les API[1] natives le code commun. Certaines limitations sont possibles, mais moins fréquentes qu'avec le multiplateforme « web ». Il existe beaucoup d'outils de ce type, dans un grand nombre de langages différents ; nous en verrons quelques-uns dans cet ouvrage. Utilisant des langages « classiques », il est ainsi

[1] *Application Programming Interface* : interface de programmation applicative. Liste les fonctions/classes/méthodes que le programmeur peut appeler pour développer ses programmes. Approchant : bibliothèque logicielle.

possible de récupérer du code d'une application « classique » (bureau) ou d'avoir des parties communes avec d'autres applications, qu'elles soient sur ordinateur, sur web ou sur mobile.

La vitesse d'exécution est proche voire identique aux applications natives (langages compilés) ; le même code peut servir à plusieurs plateformes ; utiliser un langage déjà maitrisé gagne du temps ; dans certains cas, le code peut être commun avec celui d'une application de bureau ou d'une application web.



L'aspect peut être légèrement différent des contrôles standards de la plateforme ; les outils de déploiement sont parfois moins efficaces que pour l'environnement natif. Certaines fonctionnalités ne sont pas accessibles.

Cette solution est souvent la plus intéressante et est de plus en plus utilisée par les éditeurs de logiciels à destination des téléphones. C'est la méthode qui sera le plus utilisée dans ce livre, et c'est, vous vous en doutez, celle que je préfère.

Parmi les outils multiplateformes natifs :

- **Xamarin.Forms**. Permet de coder en C#/.NET sur un PC/Windows ou un Mac. Cibles possibles : Windows UWP, Android, iOS.
- **Qt**. Permet de coder en C++ sur un PC/Windows, un PC/Linux ou un Mac. Cibles possibles : Windows, Linux, Mac OS, Android, iOS, Windows UWP.
- **Windev mobile**. Permet de coder dans un langage spécifique (W-langage) sur un PC/windows. Cibles possibles : Windows, Windows phone, Windows UWP, Linux, Android, Mac OS, iOS. Produit payant (mais une version gratuite, limitée, existe).
- **Delphi**. Permet de coder en Pascal sur un PC/Windows. Cibles possibles : Windows, Android, iOS, Mac OS, UWP. Produit payant mais il existe depuis peu une version gratuite.
- **C++ Builder**. Similaire à Delphi (même éditeur, souvent livrés ensemble) mais avec le langage C++.

Il est important de noter que dans tous les cas, quel que soit l'outil, il est **impossible** de créer une application pour iOS sans un Mac et il est impossible de créer une application pour Windows UWP sans un PC/Windows 10. Seul Android n'est pas sectaire !

Les éditeurs souhaitant développer pour les trois plateformes les plus importantes (Android, iOS, Windows 10) doivent donc utiliser au moins deux machines différentes :

- Un Apple Mac avec un système OS X 10.8 minimum
- Un PC Windows 10

4. Choisir ses outils

a) Les outils par plateforme

Pour bien choisir son outil, il convient en premier lieu de connaître sa plateforme de développement, c'est-à-dire la machine sur laquelle on souhaite écrire l'application. Celle-ci n'est évidemment pas la même que la machine qui va exécuter l'application ! On ne code pas sur un téléphone (même si les claviers virtuels ont fait des progrès) !

Le tableau suivant vous permet de savoir quels outils sont utilisables sur votre machine (Windows, macOS ou Linux) :

Tableau 3 : outils par machine de développement

	Visual Studio	Windev	Android Studio	XCode	Qt	Cordova	Xamarin	Delphi
(Windows)	Oui	Oui	Oui	Non	Oui	Oui	Oui	Oui
(Apple)	Oui	Non	Oui	Oui	Oui	Oui	Oui	Non
(Linux)	Non	Non	Oui	Non	Oui	Oui	Non	Non

Comme on le voit sur ce tableau, aucun système n'exécute tous les outils présentés. Parmi ceux-ci, Android Studio, Qt et Cordova existent sur tous les systèmes présentés.

b) Les outils par cible

Une autre manière de choisir son outil est de s'intéresser à la **cible** : sur quel téléphone voulez-vous déployer / exécuter votre application ? Il existe un grand nombre de systèmes, même si actuellement seuls trois (et encore) sont encore en activité : Android, iOS, Windows. Le cas de Windows est un peu particulier car ce n'est pas qu'un système mobile mais également pour tablettes, ordinateurs et consoles de jeu.

Le tableau suivant vous indique les cibles de chaque outil de développement pour téléphone :

Tableau 4 : Outils classés par plateforme cible

	Visual Studio	Windev mobile	Android Studio	XCode	Qt	Apache Cordova	Xamarin	Delphi/C++ Builder
	Oui	Oui	Oui	Non	Oui	Oui	Oui	Oui
	Oui	Oui	Non	Oui	Oui	Oui	Oui	Oui
	Oui	Oui[1]	Non	Non	Oui	Oui	Oui	Oui

5. Les outils par langage

Enfin, vous pouvez choisir votre outil de développement en fonction de votre langage préféré ! En effet, si vous maitrisez déjà un langage de

---

[1] Dans la version payante uniquement.

programmation, il est dommage de devoir en apprendre un autre juste pour s'essayer à la programmation sur téléphones…

La plupart des outils imposant un seul langage, un tableau n'est pas intéressant. Voici une liste (non exhaustive) d'outils permettant de programmer pour téléphone ou tablette dans votre langage préféré :

- C++ : Qt, C++ Builder, Visual Studio
- C# : Visual Studio
- Pascal : Delphi
- Python : Kivy
- HTML/CSS/Javascript : Cordova
- Java : Android Studio
- Objective-C, Swift : XCode
- Ruby : XCode

6. Les outils utilisés

Pour résumer rapidement le choix de l'outil à utiliser, voici une petite liste des outils présentés dans ce livre avec leur « plus » :

- Le plus *Android* : Android Studio
- Le plus *iOS* : XCode
- Le plus *Windows* : Visual Studio
- Le plus *open source* : Qt
- Le plus efficace : Xamarin
- Le plus polyvalent : Cordova

Pour réaliser cet ouvrage, j'ai utilisé tous ces outils, sur différentes machines. A l'heure où vous lisez ces lignes, il est fort possible que des versions plus récentes de ces outils existent et qu'elles soient légèrement différentes des outils employés, mais les différences seront mineures.

- Système **Windows 10** (version 1803 dite « *april update* »)
    - Visual Studio 2017 (version 15.8)
    - Qt 5.11
    - Android Studio 3.1.2
- Système **macOS** (version 10.11 dite « *El Capitan* » et version 10.14 dite « Mojave »)
    - Visual Studio 2017 (version 15.6)

   - XCode 8.1 (sous 10.11) et 9.2 (sous 10.14)
   - Android Studio 3.1.2
   - Qt 5.9
- Système **KUbuntu** (version 18.04)
   - Android Studio 3.1.2
   - Qt 5.10

Les tests ont été réalisés sur les périphériques suivants (en plus des émulateurs logiciels) :

- Téléphone Microsoft Lumia 950xl sous Windows 10 (version 1709)
- Tablette Samsung Galaxy Tab 2 sous Android 4.1
- Téléphone Nokia 5 sous Android 8.1
- Téléphone Apple iPhone 4 sous iOS 9.3
- Tablette Boulanger sous Windows 10 (version 1803)

En plus de ces outils, spécifiques au développement, j'ai utilisé :

- *TortoiseSVN* (sur PC/Windows 10) couplé à un serveur SVN/Apache personnel, pour la gestion de version des codes sources
- *SmartSVN* (sur macOS) couplé au même serveur
- *Frama.link*, outil web permettant d'obtenir des URL raccourcies
- *Créateur d'icônes*, un outil personnel permettant de créer facilement des jeux d'icônes de taille standard pour les applications Android, iOS, UWP. Disponible sur le Windows Store.
- Microsoft Office pour la rédaction de ce livre
- Pencil pour la réalisation des maquettes d'IHM
- 7-zip pour la création des archives
- XAMP pour un serveur web local, avec PHP, pour tester les services web

IV. Outils de développement

Il existe de nombreux outils de développement, et un grand nombre d'entre eux permettent de faire du développement pour téléphones. Nous allons dans ce chapitre examiner quelques-uns de ces outils, décrire rapidement leur installation et réaliser une petite application (la même application pour chaque outil) afin de tester l'installation et d'appréhender l'outil. Certains outils sont multiplateformes, peuvent utiliser plusieurs langages, tandis que d'autres sont réservés à une seule plateforme. La liste n'est pas du tout exhaustive mais se veut suffisamment complète pour permettre de choisir efficacement son outil de développement. Bien entendu, si vous ne souhaitez découvrir qu'un seul outil, vous pouvez ne lire que les parties lui étant consacrées.

Quand on découvre un nouveau langage, la tradition des programmeurs est de faire ce qu'on appelle un « *hello world* », un petit programme qui se contente d'afficher « *hello world* » (bonjour le monde) à l'écran. Nous allons aller un peu plus loin : notre programme va bien afficher « *hello world* » (c'est la tradition) mais également placer sur l'écran un bouton et afficher le nombre de clics effectués par l'utilisateur sur ce bouton. Ce petit programme permettra donc de tester à la fois l'affichage et l'interaction avec l'utilisateur. J'aime bien appeler un tel programme un « *hello world, clic me !* » (Bonjour le monde, clique moi !).

1. Android Studio

Android Studio est l'outil de développement natif pour la plateforme Android. Vous ne pouvez pas écrire d'application pour iOS ou Windows avec cet outil ! A n'utiliser que si vous êtes sûrs de ne faire que de l'Android.

Android Studio est un EDI[1] basé sur *IntelliJ Idea* (un EDI Java). Il existe pour un grand nombre de systèmes d'exploitation, notamment Windows, Linux et mac OS. Il intègre tous les outils nécessaires au développement Android, notamment le SDK[2]. Le langage utilisé pour coder l'application est

---

[1] Environnement de Développement Intégré (IDE en anglais) : logiciel intégrant tous les outils nécessaires au développement d'application : éditeur, compilateur, débogueur…

[2] *Software Development Kit* : kit de développement logiciel. Ensemble d'outils logiciels destinés au développeur de logiciels.

classiquement le Java, mais récemment Google a ajouté le langage *Kotlin* pour le développement Android. L'EDI est identique sur chacune des plateformes supportées.

a) Installation d'Android Studio

Pour installer Android Studio il suffit de se rendre sur cette page et de télécharger l'outil : https://developer.android.com/studio/index.html.

Exécutez l'installateur et suivez les différentes étapes proposées.

Si vous n'avez pas déjà de SDK Android, installez-le en même temps qu'Android Studio ; dans le cas contraire, il faudra spécifier le chemin de votre installation du SDK.

Il n'y a pas d'autre option à l'installation (à part bien entendu le chemin de celle-ci), il ne reste qu'à patienter pendant que celle-ci se termine !

b) Petite application

Testons l'installation d'Android Studio avec une petite application « Hello world, clic me ».

Exécutez Android Studio, puis choisissez « *Start a new android studio project* », donnez-lui un nom et choisissez-lui un nom de domaine de compagnie (peu utile pour un particulier) et le chemin du projet. La fenêtre suivante permet de choisir la cible : téléphone ou tablette, vêtement intelligent, montre, TV… ainsi que le niveau minimum du SDK Android. Ce choix est très important : si vous choisissez un niveau faible, vous pouvez cibler beaucoup d'appareils Android mais certaines fonctionnalités n'existeront pas. Si en revanche vous prenez un niveau fort (une API récente), vous ciblerez moins d'appareils mais vous aurez plus de fonctionnalités… Pour cette petite application le niveau d'API 15 est suffisant.

La fenêtre suivante permet de créer une activité (un écran) parmi plusieurs styles d'activités différentes. Le plus simple est de choisir « *empty activity* ». Une fenêtre s'ouvre dans laquelle on définit le nom des deux composants d'une activité : le *layout* et la classe Java.

Un *layout* est un fichier de description d'un écran, basé sur le langage XML, et la classe Java correspondante associée permet de lier du code à la

fenêtre. Chaque écran Android (chaque activité) comprendra une classe Java associée (héritant de la classe `Activity`) et, souvent, un *layout*.

L'assistant crée ensuite les fichiers nécessaires, ce qui prend un peu de temps ! Il faut être patient avec Android Studio…

Le *layout* peut être édité soit directement en XML (ce qui demande de connaître un peu les composants Android visuels), soit visuellement par glisser-déposer.

Pour notre petite application, nous aurons besoin de zones de texte (`TextView`), et d'un bouton (`Button`). Les activités Android sont basées sur un système de conteneurs visuels qui permettent d'aligner automatiquement les composants, soit linéairement (`LinearLayout`), soit dans une grille (`GridLayout`), soit par rapport les uns aux autres (`RelativeLayout`).

Ici pour faire simple, un conteneur linéaire vertical est suffisant. Plaçons ensuite à l'intérieur les deux composants `TextView` et le bouton de la manière suivante :




```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_alignParentBottom="true"
    android:layout_alignParentRight="true"
    android:layout_alignParentEnd="true"
  xmlns:android="http://schemas.android.com/apk/res/android">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World !"
        android:id="@+id/textView"
        android:layout_gravity="center_horizontal"
        android:textSize="28sp" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```
        android:text="Clic me !"
        android:id="@+id/button"
        android:layout_gravity="center_horizontal"
        android:textSize="24sp" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"

     android:textAppearance="?android:attr/textAppearanceLarg
     e"
        android:text="0 clic(s)"
        android:id="@+id/texte"
        android:layout_gravity="center_horizontal" />
</LinearLayout>
```

Quelques petites explications sur les propriétés de ce fichier XML…

Chaque composant a une propriété `android:id` qui sert à l'identifier. Cette propriété est une chaîne de caractères. Le symbole `@+id` indique que l'identifiant est nouveau, son nom suit le / (par exemple `@+id/texte` indique que le nom du contrôle est `texte`). Ce nom doit être unique dans le *layout*.

Les `TextView` ont une propriété `android:text` qui est le texte affiché, une propriété `android:textSize` qui permet de régler la taille du texte (ici les tailles sont données en `sp`, c'est-à-dire en taille relative de police).

Parmi les propriétés importantes, on trouve `layout_width` qui gère la façon dont le contrôle se comporte dans son conteneur parent, par exemple est-ce qu'il occupe toute la largeur (`layout_width="match_parent"`), s'il occupe la place minimale suivant son contenu (`wrap_content`) ou s'il prend une valeur numérique directe (`20 px` par exemple). La propriété `layout_height` est similaire pour la hauteur.

Une fois ce *layout* créé, il faut éditer la classe Java correspondante.

La première chose à faire est d'ajouter des attributs dans la classe, correspondants aux composants graphiques qui doivent être manipulés dans le code, ici le `TextView` nommé `texte` et le bouton.



```
private Button bouton;
```

```
    private TextView texte;
```

Il importe ensuite d'initialiser ces attributs, grâce à la fonction `onCreate`, présente dans toutes les activités. Pour initialiser les attributs, on utilise la fonction `findViewById` et l'identifiant déclaré dans le *layout* :

```
bouton = (Button)findViewById(R.id.button);
texte = (TextView)findViewById(R.id.texte);
```

La fonction `findViewById` prend en paramètre un entier qui est un numéro unique par composant. Ces numéros sont définis dans la classe `R` (pour Ressources), dans sa sous-classe `id` (contenant les identifiants). Cette fonction renvoyant un `Object`, il faut donc le transtyper[1]…

Il est ensuite nécessaire de relier une fonction au clic sur un bouton… pour cela il faut une classe qui implémente l'interface `OnClickListener` : notre activité va donc l'implémenter, ce qui lui oblige à avoir une opération `onClick`. On relie le bouton à l'activité, toujours dans la fonction `onCreate` :

```
bouton.setOnClickListener(this);
```

Après l'exécution de cette fonction, à chaque clic de l'utilisateur sur le bouton, l'opération `onClick` de l'activité sera appelée. Nous allons donc placer dedans le code nécessaire pour compter le nombre de clics, à savoir un attribut de type entier, initialisé à 0 pour compter les clics :

```
private int compteur=0;
```

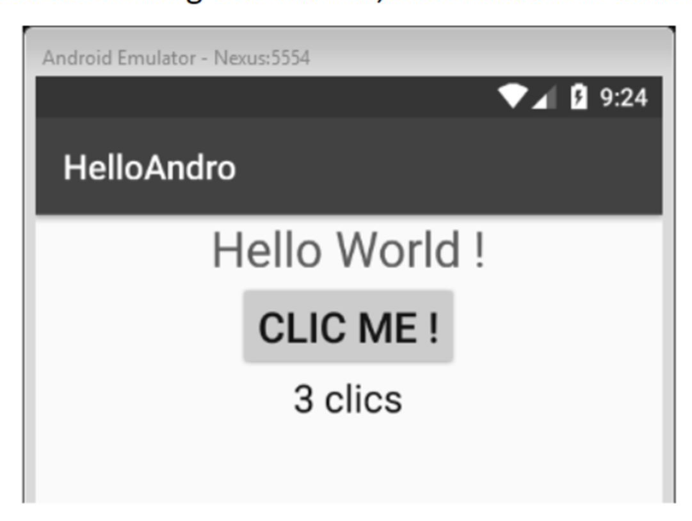
Et enfin, l'opération `onClick` devient tout simplement :

```
@Override
    public void onClick(View view) {
        ++compteur;
        texte.setText(String.valueOf(compteur)+" clics");
    }
```

[1] Transtyper : changer le type d'un objet. L'objet doit être d'un type polymorphe et être compatible avec le type cible.

Pour tester, vous pouvez soit brancher un périphérique Android (smartphone, tablette…), soit utiliser l'émulateur.

L'émulateur est assez long à démarrer, alors laissez-le ouvert !



Capture 1 : Hello world Android avec Android Studio

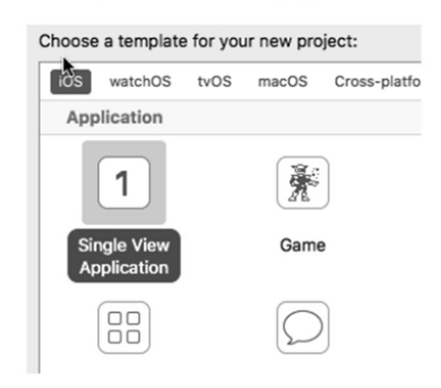
2. XCode

Comme vu précédemment, l'environnement XCode n'existe que sous macOS. Pour développer pour iPhone, il faut au minimum la version 4 de XCode et 10.6 de macOS. Pour utiliser le langage *Swift*, il faut au minimum la version 6 de XCode et 10.10 de macOS (sinon il faut se contenter du langage *Objective-C*). Pour pouvoir publier sur l'Apple Store, il faut la version 9 de XCode pour pouvoir utiliser le SDK iOS 11, donc la version 10.12 (Sierra) de macOS. Attention : ceci est valable à l'heure où ces lignes sont écrites, il est fort probable que dans le futur les versions minimales soient plus récentes : il faut donc garder son Mac et XCode à jour !

a) Installation de XCode

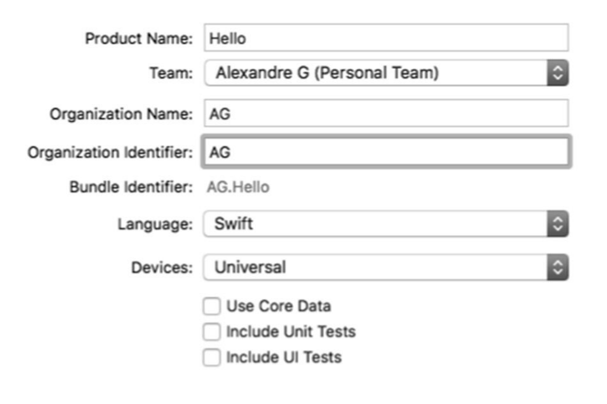
L'installation se fait simplement à partir de l'Apple Store. Attention, seule la dernière version de XCode peut être téléchargée ainsi, et celle-ci nécessite la dernière version de macOS. Si vous voulez installer une version plus ancienne, il faut la télécharger sur le site d'Apple consacré aux développeurs. Un compte développeur (même gratuit) est alors nécessaire. L'installation est quasiment automatique et très simple à effectuer.

b) Petite application



Exécutez XCode. Dans l'écran d'accueil choisissez « *Create a new Xcode project* », puis un projet de type « iOS », « *single view application* ».

Déterminez un nom pour le produit, rentrez votre compte développeur Apple qui sera lié à votre équipe de développement, choisissez un nom d'organisation (la combinaison *Name/Identifier* doit être unique), prenez le langage *Swift* et le mode *Universal* comme sur la capture ci-dessous :



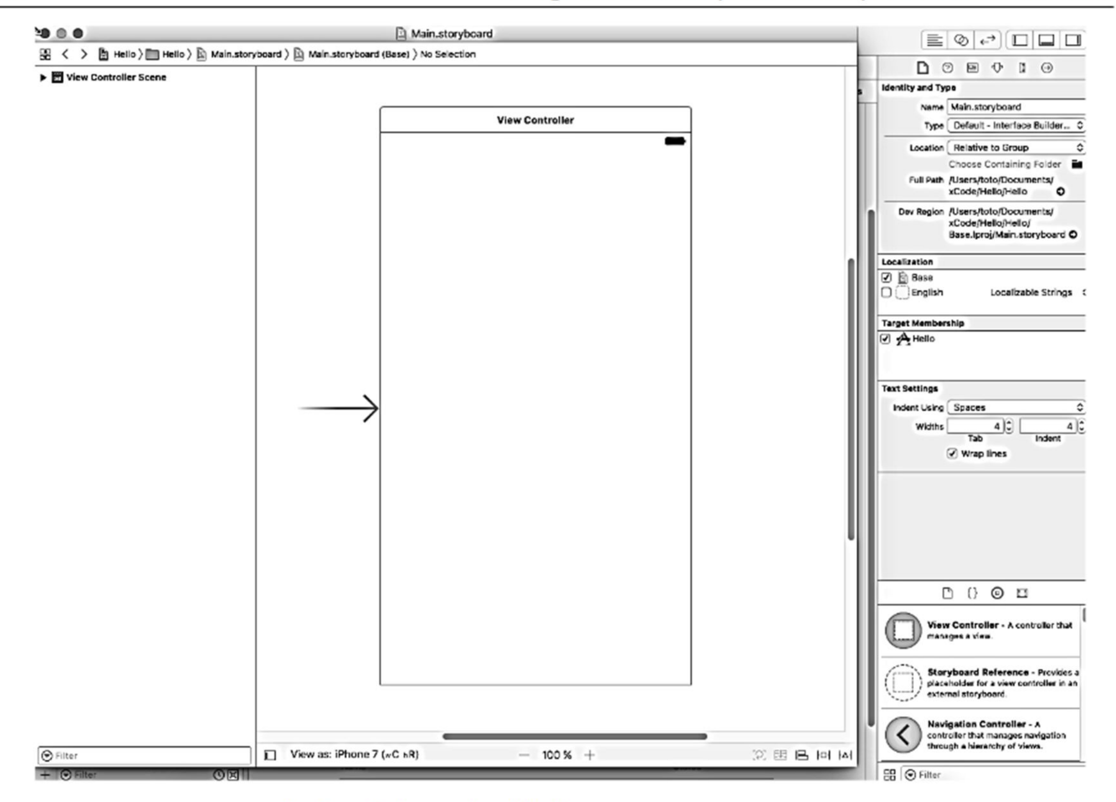
Capture 2 : création d'un projet XCode

Il faut ensuite indiquer un emplacement pour le projet (par défaut dans Documents) et choisir *Create*.

XCode va ensuite s'ouvrir sur la vue principale. Dans le volet de gauche, on remarque la structure globale de l'application, avec notamment :

- Le fichier `main.storyboard` qui contient la description de l'écran principal
- Le fichier `ViewController.swift` qui contient le code du contrôleur de l'écran principal

Commençons par éditer le *storyboard*, autrement dit la vue de l'écran principal de l'application. Double-cliquez sur `main.storyboard` dans la vue de gauche pour éditer visuellement celui-ci. Vous obtenez une vue d'un écran vierge au centre, une vue arborescente du contenu de l'écran à gauche, et un ensemble d'outils à droite :

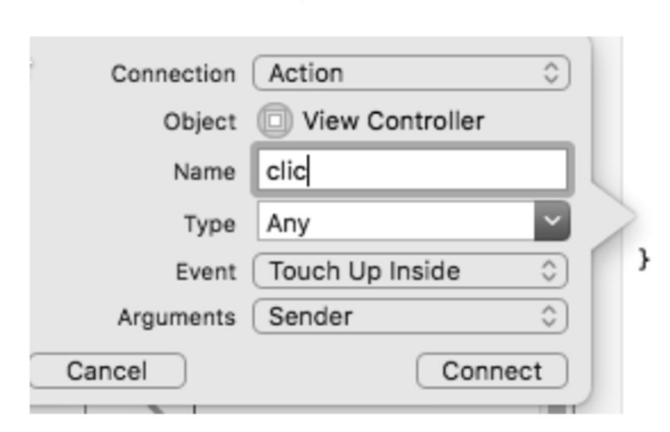


Capture 3 : la vue dans XCode

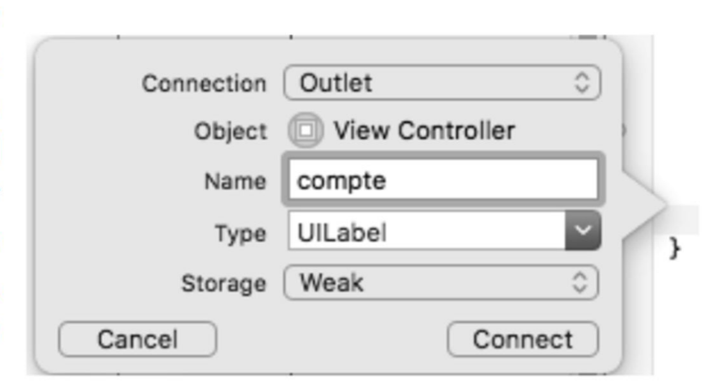


En bas à droite, les contrôles visuels sont disponibles et peuvent être glissés-déposés sur la vue : placez de cette manière un `Label`, un bouton et un autre `Label`:

Il faut ensuite répondre au clic sur le bouton ; pour cela, placez-vous en vue étendue en cliquant sur ce bouton : . La page du code du contrôleur apparait à droite de la vue. Pour lier le clic sur le bouton au code, glissez-déplacez le bouton **en laissant la touche CTRL enfoncée** dans la vue. Une fenêtre apparait alors pour configurer le lien (ici type `Action`).



Une fonction est alors créée dans le code. Nous allons y placer le code nécessaire à l'affichage du nombre de clics, mais auparavant il faut relier le *label* qui va l'afficher au code. Pour cela, procédez de la même manière que pour le bouton, mais cette fois avec une connexion *outlet* :



Un attribut de la classe est alors rajouté automatiquement :

```swift
@IBOutlet weak var compte : UILabel!
```

Il nous faut ajouter un attribut pour le comptage :

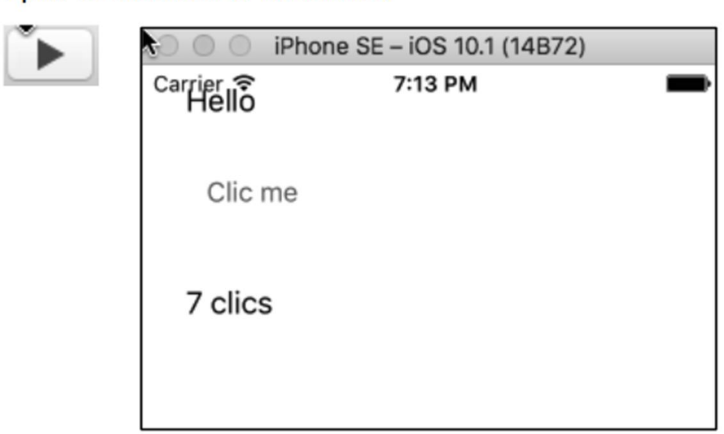
```swift
private var compteur=0
```

Il ne reste qu'à compléter la fonction de réponse au bouton précédemment créée :

```swift
@IBAction func clic(_ sender : Any)
{
     compteur = compteur+1
     compte.text = String(compteur)+" clics"
}
```

Swift

Choisissez le type de périphérique simulé (par exemple, iPhone SE) ou branchez en USB votre propre iPhone, puis lancez la compilation puis l'exécution par le bouton ci-dessous.



Capture 4 : "Hello world" avec XCode sur simulateur iPhone

3. Visual Studio

L'EDI Visual Studio peut être utilisé, en programmation mobile, avec trois environnements différents :

- Xamarin (intégré) permet de faire du développement Android, Windows et iOS en utilisant C# et le *framework* .NET
- Cordova (intégré) permet de faire du développement Android, Windows et iOS en utilisant les langages du web : HTML, CSS, Javascript.
- UWP (intégré) permet de faire du développement pour Windows 10 uniquement, quelle que soit la plateforme (mobile, bureau, Xbox, etc…)

Grâce à l'outil Xamarin, il est possible d'utiliser Visual Studio, le langage C# (mais aussi C++, VB, F#...) et le *framework* .NET pour écrire des applications pour téléphones. L'outil est disponible pour Windows et macOS (pas de Linux disponible pour l'instant).

La version Windows peut faire des applications pour Windows (classique), Windows 10 (universel) et Android.

La version macOS, moins complète, limitée au langage C#, peut réaliser des applications pour macOS, iOS et Android.

L'outil Cordova (voir III.2), quant à lui, permet d'utiliser les langages « du web » comme HTML/CSS et Javascript pour développer une application sur les différentes plateformes mobiles. Cordova n'est intégré qu'avec la version Windows de Visual Studio. En sus de Javascript, il est possible d'utiliser *Typescript*, langage proche mais plus « propre » développé par Microsoft, avec Visual Studio.

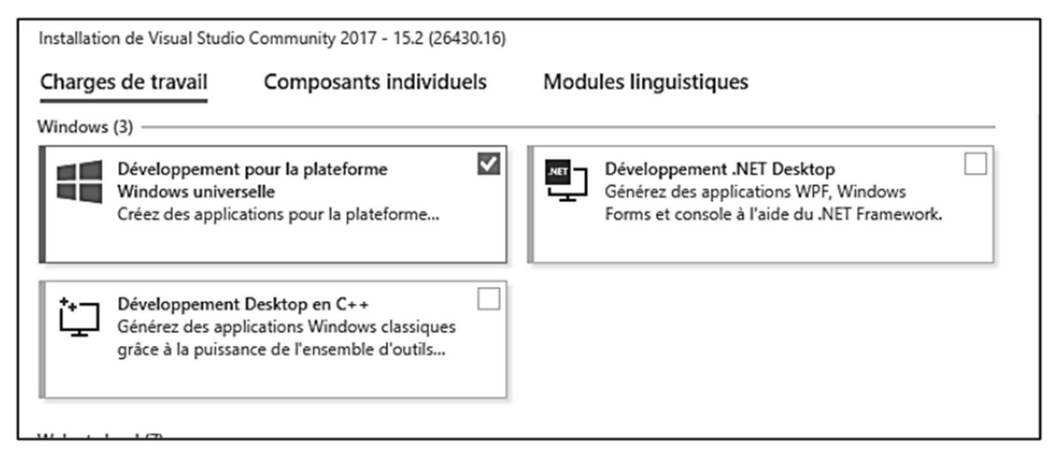
Il existe une version gratuite de l'outil, *Visual Studio Community*, dont les fonctionnalités sont suffisantes. Je vous invite à bien consulter la licence pour connaître les conditions d'utilisation.

Avec Xamarin (voir page 43), le langage utilisé ici est C#, langage le plus adapté au *framework* et sera le seul utilisé avec cet outil dans cet ouvrage. Xamarin est à la fois natif et multiplateforme, et grâce à différents outils (comme *Xamarin.Forms* pour l'interface utilisateur, et *Xamarin.Plugins*) il est souvent possible de faire un code 100% portable entre les différents systèmes mobiles.

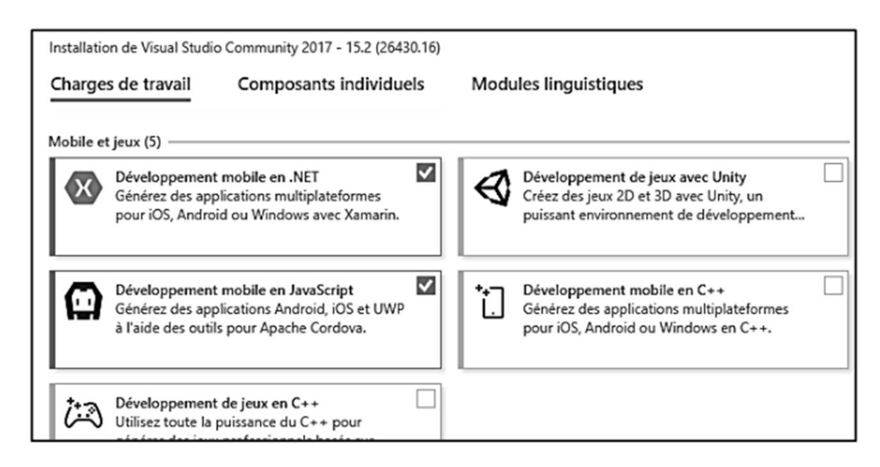
a) Installation de Visual Studio 2017 community

Avec un PC/Windows ou un Mac, l'outil peut se télécharger à l'adresse suivante : https://www.visualstudio.com/fr/vs/community.

Le téléchargement ne correspond qu'à une petite application servant à installer l'ensemble ; suivant les options choisies, l'installation peut être très importante. Visual Studio peut en effet être utilisé pour de nombreux développements ! Ceux-ci sont découpés en charges de travail, avec des composants facultatifs.

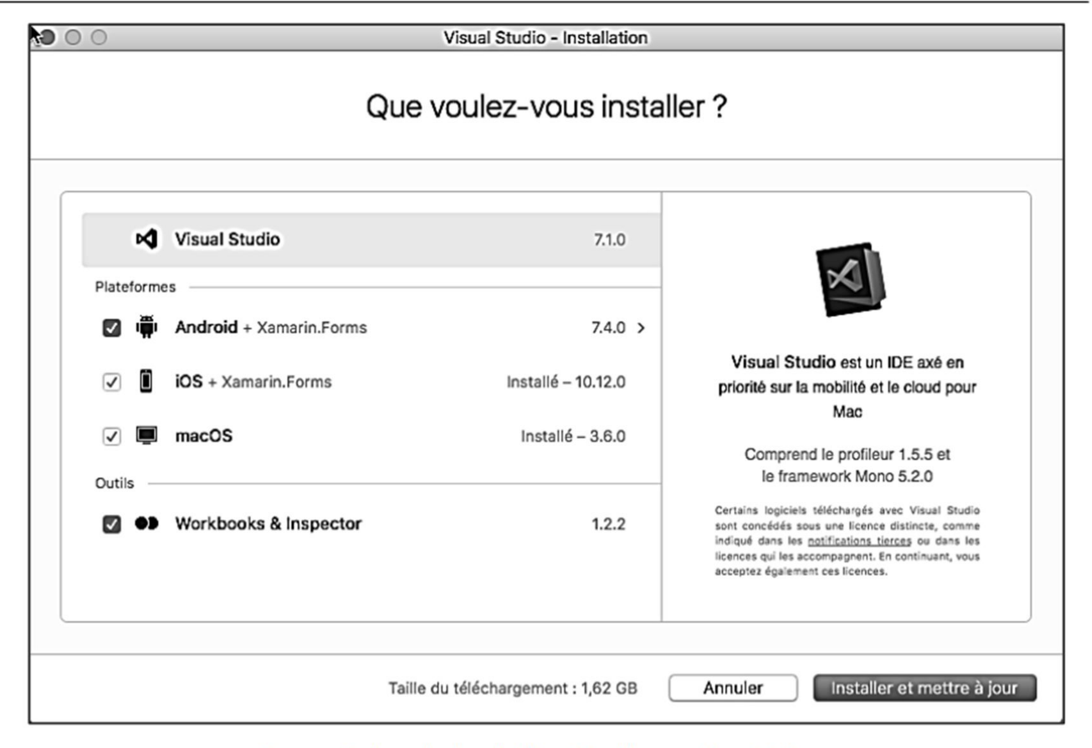


Cochez « développement pour la plateforme windows universelle » pour faire du développement UWP (Windows 10 mobile)



Cochez « développement mobile en .NET pour inclure Xamarin ; les composants facultatifs comme le SDK Android ou le JDK peuvent ne pas être inclus si vous les avez déjà. Cocher « développement mobile en C++ » si vous voulez pouvoir faire du C++.
Cochez « développement mobile en JavaScript » pour inclure Cordova.

Armez-vous de patience : l'installation va être assez longue, surtout si votre connexion Internet n'est pas très rapide.



Capture 5 : installation de Visual Studio sous Mac OS X

b) Xamarin pour iOS avec un PC

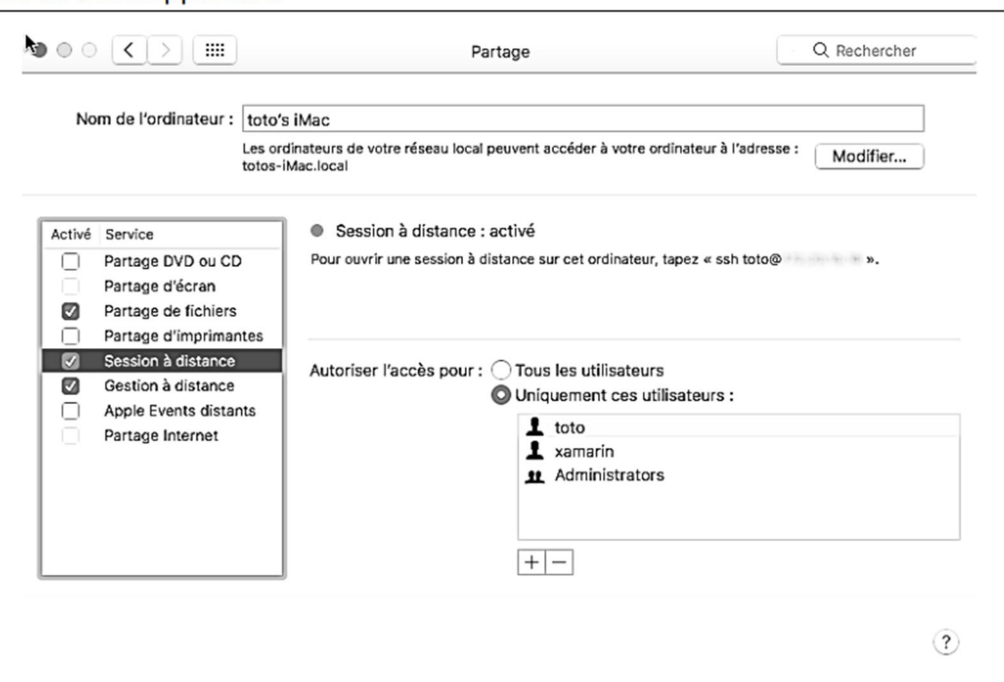
Pour pouvoir utiliser *Xamarin.iOS* et donc réaliser des applications pour iPhone/iPad, il faut impérativement un ordinateur Apple avec le système macOS correctement installé et configuré.

Si vous travaillez avec Visual Studio directement sous le Mac, vous n'avez rien de spécial à faire ; en revanche, si vous souhaitez travailler sur un PC, il faut tout de même avoir un Mac sur votre réseau (même si peu puissant), ce Mac devant être accessible par votre PC.

Installez et mettez à jour sur le Mac XCode, le SDK iOS, Visual Studio avec *Xamarin*. Faites attention à bien avoir les versions les plus récentes, et surtout la même version de *Xamarin* sous Mac et sous PC.

Sur le PC, installez ou mettez à jour Visual Studio et *Xamarin*.

Il faut à présent vérifier que le Mac et le PC peuvent se « parler » en réseau. Sur le Mac, il faut activer le partage réseau : préférences systèmes → Partage → activez le paramètre « Session à distance » pour autoriser l'accès SSH. Relevez le nom et l'adresse IP de votre Mac.



Capture 6 : configurer la session à distance sous macOS

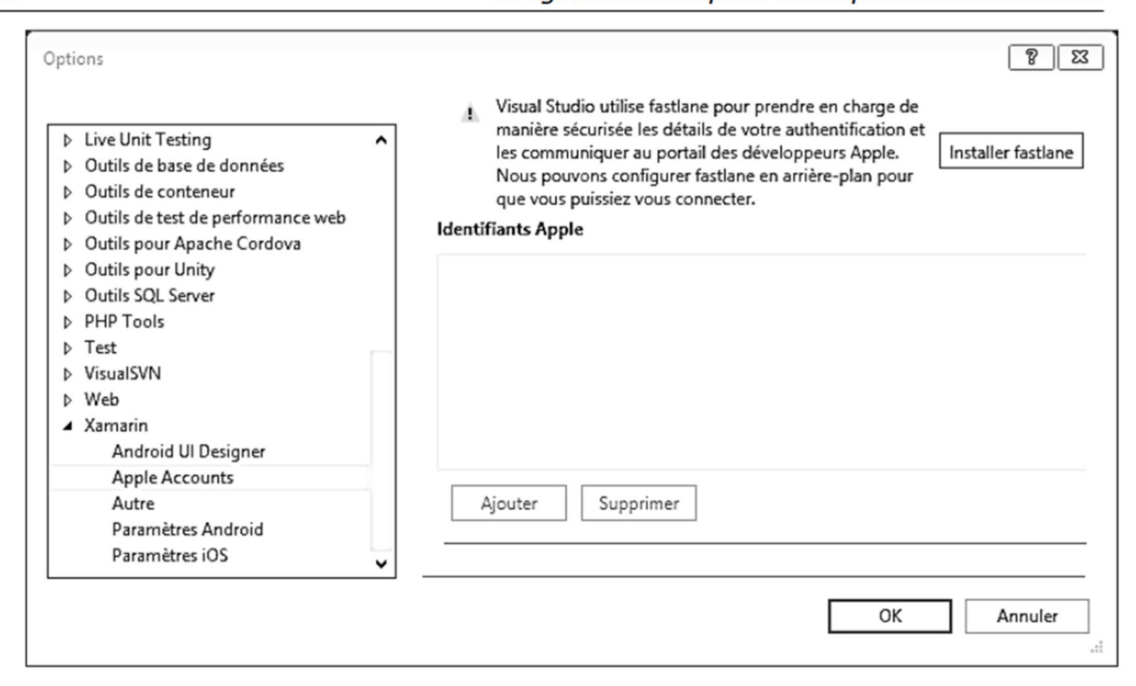
Sur le PC, dans Visual Studio, configurez l'agent Xamarin par le menu « outils → iOS → Agent Mac Xamarin ». Cliquez sur « ajoutez un serveur » si votre Mac n'est pas automatiquement détecté (normalement, c'est le cas) puis sur « connexion » : les identifiants du compte Mac utilisé seront demandés.

Si toutes ces manipulations semblent trop complexes, il est possible de copier votre projet Xamarin depuis votre PC sur votre Mac (avec le réseau ou une clé USB) et d'utiliser directement Visual Studio sur le Mac, mais cela est moins pratique (sauf si vous ne développez que pour iOS évidemment).

Pour déployer son application sur un vrai périphérique (iPhone, iPad ou iPod) il faut :

- Un compte développeur Apple (même gratuit)
- Autoriser l'appareil à exécuter des applications ne venant pas de l'Apple store
- « Provisionner » son appareil pour l'application à déployer.

Dans Visual Studio, allez au menu *outils→options→Xamarin→Apple Accounts*.



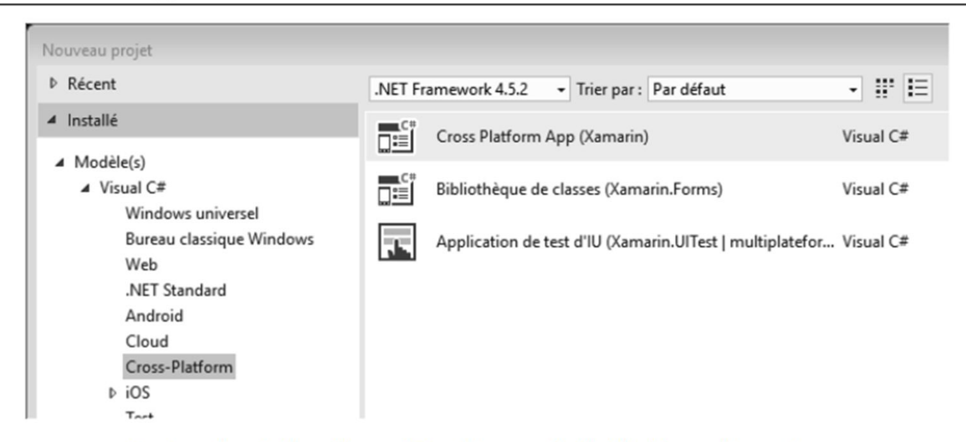
Installer *fastlane* si ce n'est pas déjà fait (les identifiants du Mac connecté sur le réseau seront demandés). Faites ensuite ajouter un identifiant Apple (votre compte développeur Apple). Ouvrir ensuite les options du projet iOS, onglet « signature du bundle iOS ». Choisir le schéma « automatique » puis l'équipe de développement dans la zone déroulante « provisionnement automatique ».

Vous pouvez également utiliser XCode sur le Mac pour l'associer à un compte développeur. Sur le PC, dans Visual Studio, il sera possible d'utiliser ce compte pour « provisionner » l'application automatiquement.

Il faut ensuite créer l'application XCode, l'associer avec le compte développeur, générer la signature automatique (*automatically manage signin*g) et utiliser le même identificateur (*Bundle Identifier*) dans le projet Xamarin.

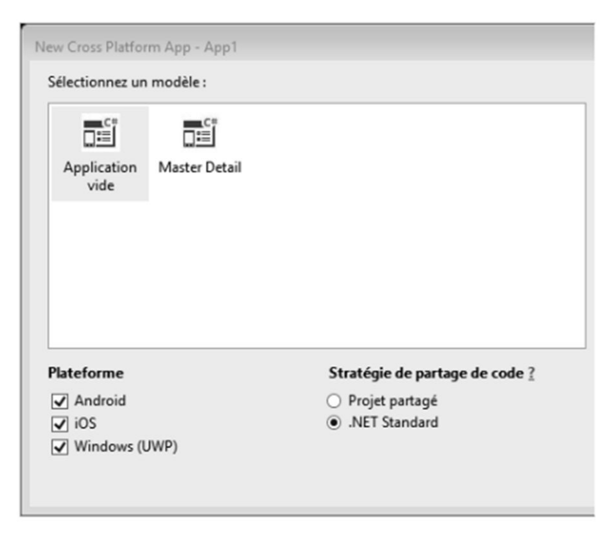
c) Petite application (Xamarin)

Testons l'installation avec une petite application « *hello world, clic me !* ». Une fois Visual Studio exécuté, il faut créer un projet soit Android, soit iOS, soit Windows universel, soit « *Cross-platform* », c'est-à-dire un projet commun à plusieurs plateformes.



Capture 7 : création d'un projet mixte pour Hello World avec Xamarin

Avec Xamarin, il y a plusieurs choix possibles sur l'écran de création de solution suivant :



Capture 8 : paramètres nouveau projet Xamarin

Le modèle permet de choisir le type d'application que l'on veut ; ici nous prendrons « application vide », le modèle le plus simple. Vous pouvez choisir la/les plateforme(s) de votre projet. La stratégie de partage de code indique comment le code partagé est réparti : soit dans un projet partagé (compilé sur chaque plateforme), soit par du code .NET totalement indépendant de la plateforme réelle. Sur une application aussi simple, cela n'a pas d'importance, nous allons choisir, de façon quasi-systématique, « .NET Standard ».

L'assistant crée plusieurs projets :

- Un projet partagé. Le projet doit être totalement multiplaforme et ne peut comporter que du code commun. Les parties visuelles peuvent être placées dans ce projet avec *Xamarin.Forms*.
- Un projet pour Android. Le projet représente l'application Android. Il va contenir le code spécifique à Android, les ressources visuelles spécifiques, etc.
- Un projet pour iOS qui contient le code pour iOS. Ce projet n'est pas compilable sur un PC, il faut un Mac relié au même réseau.
- Un projet pour Windows Universel qui contient le code spécifique à Windows 10.

Il faut en premier lieu éditer la partie visuelle, l'interface utilisateur : le fichier `MainPage.xaml`. Avec *Xamarin.Forms*, celle-ci est commune à toutes les plateformes, mais hélas ne peut pas être éditée visuellement, avec un éditeur WYSIWYG[1] : il faut l'éditer en XAML[2] directement. Si nous avions choisi une IU[3] native, l'édition visuelle serait possible, mais il faudrait en faire une pour Android, une pour iOS, une pour Windows. Il existe un générateur d'aperçu pour *Xamarin.Forms* XAML, accessible dans le menu « Affichage→Autres fenêtres, *Xamarin.Forms previewer* ». Il n'est pas encore totalement finalisé et peut connaitre des petits soucis et ralentissements, mais il permet de voir rapidement à quoi ressemble l'interface saisie en XAML sans avoir besoin d'exécuter l'application.

Nous allons créer une interface utilisateur très simple : une étiquette texte en haut, un bouton, une étiquette texte.

Le code XAML est le suivant :

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
     xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:HelloXamarin"
             x:Class="HelloXamarin.MainPage">
    <StackLayout>
        <Label Text="Hello world !"
```

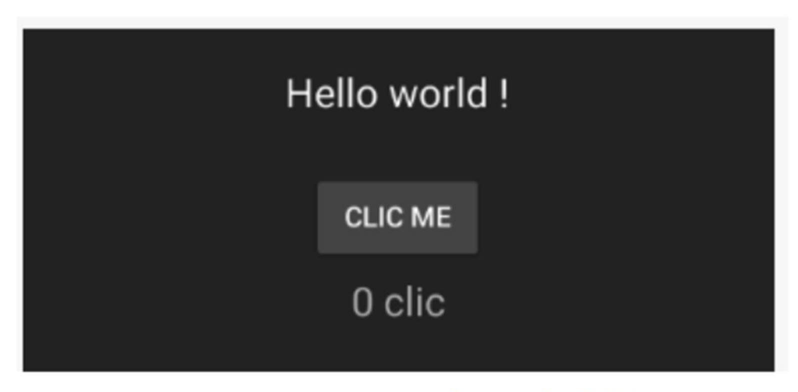
[1] What You See Is What You Get : ce que tu vois est ce que tu obtiens. Signifie que l'aspect réel sera similaire à l'aspect de l'édition.

[2] XAML est une version du langage XML adaptée à l'édition d'interface utilisateur avec Visual Studio.

[3] Interface Utilisateur.

```
            HorizontalOptions="Center"
            FontSize="20"
            TextColor="White"
            Margin="20" />
         <Button x:Name="clic" Text="Clic me"
             HorizontalOptions="Center" />
         <Label Text="0 clic" x:Name="clics"
                FontSize="20" HorizontalOptions="Center" />
    </StackLayout>
</ContentPage>
```

Ce qui donne un aperçu comme sur la capture suivante (l'aperçu fonctionne pour Android et iOS si un Mac est disponible sur le réseau) :



Capture 9 : Aperçu XAML Xamarin.Forms (Android)

Editons à présent le fichier `MainPage.cs` qui est associé au fichier XAML. Pour cela, associons à l'évènement `Clicked` du bouton une opération, tout simplement par le code suivant dans le constructeur :

```
public MainPage() {
    InitializeComponent();
    clic.Clicked += Clic_Clicked;
}
```

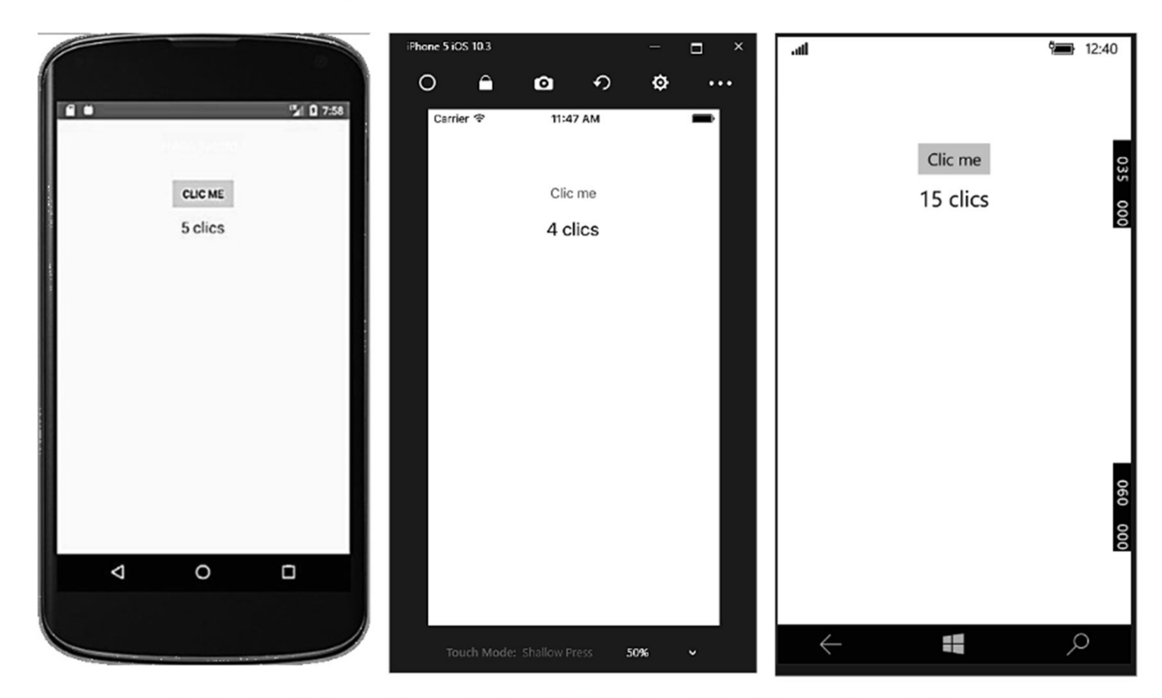
Le code lui-même de cette opération est simple :

```
private void Clic_Clicked(object sender, EventArgs e) {
    ++cpt;
    clics.Text = cpt.ToString() + " clics";
}
```

Bien sûr, il est nécessaire d'ajouter dans la classe un attribut pour réaliser le compteur de clics :

```
private int cpt = 0;
```

Il est ensuite simple de tester l'application sous Windows, Android (avec un émulateur ou un périphérique Android) ou iOS (simulateur ou iPhone, si un Mac est accessible).



Capture 10 : Hello world pour Android, iOS, Windows 10 mobile avec Xamarin Forms

Si vous souhaitez, à partir d'un PC sous Windows, compiler votre application pour iOS, il faut absolument avoir sur votre réseau local un Mac contenant :

- XCode et le SDK iOS installés (versions les plus récentes)
- Xamarin (même version que celle de Visual Studio sur PC, si possible la plus récente)

Sur le Mac, il faut autoriser les accès réseau à distance pour que Visual Studio puisse, par SSH[1], se connecter à l'agent Xamarin.

Si vous possédez un Mac mais que vous ne pouvez le contrôler depuis Visual Studio ou que vous préférez travailler depuis le Mac, vous pouvez copier le dossier source du projet Visual Studio et l'ouvrir avec Visual Studio pour Mac. Vous pourrez alors compiler et tester votre application pour iOS.

---

[1] SSH : *Secure Shell*. Système sécurisé, basé sur du chiffrage asymétrique, d'accès à distance à une autre machine (44).

d) Cordova pour iOS avec un PC

Il est possible avec Visual Studio 2017 sur un PC de développer une application avec Cordova avec un iPhone (ou iPad, iPod…) comme cible, mais pour cela un Mac, accessible sur le réseau, est obligatoire. En plus des outils XCode, il faut installer un certain nombre d'autres outils, tels que décrit dans (4).

Préparons d'abord le Mac :

Il faut installer les outils de ligne de commande de XCode, qui ne sont pas toujours présent par défaut (je pars du principe que XCode est déjà installé). Il suffit pour cela d'exécuter dans un terminal :

```
xcode-select --install
```

Ensuite, l'outil Node.js, disponible sur https://nodejs.org doit être installé (prendre la version LTS).

Ensuite il faut installer l'outil « *remote agent* » qui permet de construire à distance une application en exécutant dans un terminal (suivant la configuration des droits d'accès, vous pouvez avoir besoin de faire précéder l'instruction par `sudo`[1]) :

```
npm install -g remotebuild
```

Une fois l'installation terminée, il faut l'exécuter, en mode sécurisé ou non (sur un réseau local, inutile d'utiliser le mode sécurisé) :

```
remotebuild --secure false
```

Au premier lancement, de nombreux modules doivent être installés encore, acceptez-les. Le serveur se lance alors en indiquant le numéro du port utilisé : notez-le (3000 par défaut) !

---

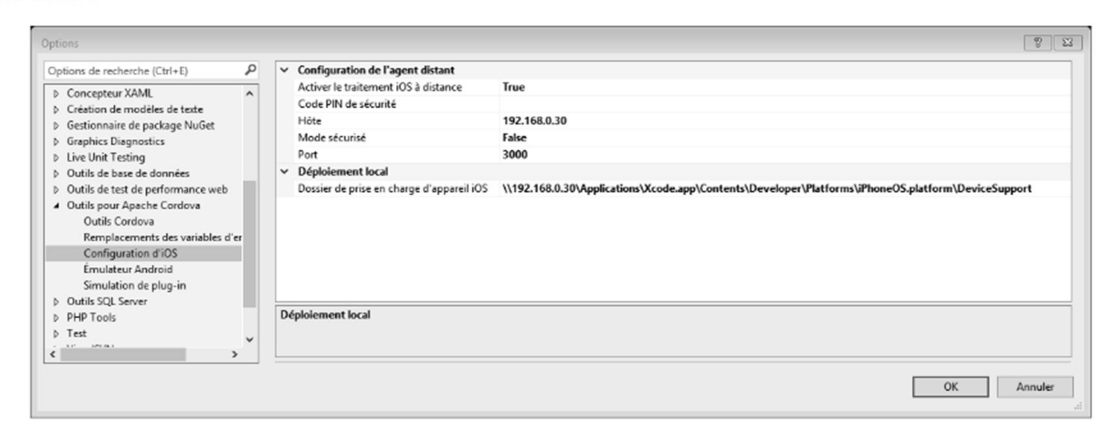
[1] sudo : sur un système Linux ou macOS, demande une élévation de droits au niveau administrateur pour la commande qui suit (il faut que le compte utilisé ait ce droit, évidemment).

```
Installation réussie des packages homebrew.

Serveur de builds distant à l'écoute sur [http], port 3000
Affichez/modifiez la configuration du serveur sur /Users/alexandre/.taco_home/Re
moteBuild.config. Vous devrez peut-être exécuter 'remotebuild saveconfig' pour l
a générer. Vous devrez redémarrer le serveur si vous mettez à jour ses configura
tions.
```

Capture 11 : installation de remotebuild sur macOS terminée

Vous pouvez maintenant passer à la configuration sur le PC, dans Visual Studio 2017 : menu « outils → options → Outils pour Apache Cordova → configuration d'iOS ». Indiquez les informations comme dans la capture suivante, en remplaçant l'adresse IP par celle de votre MAC sur votre réseau :

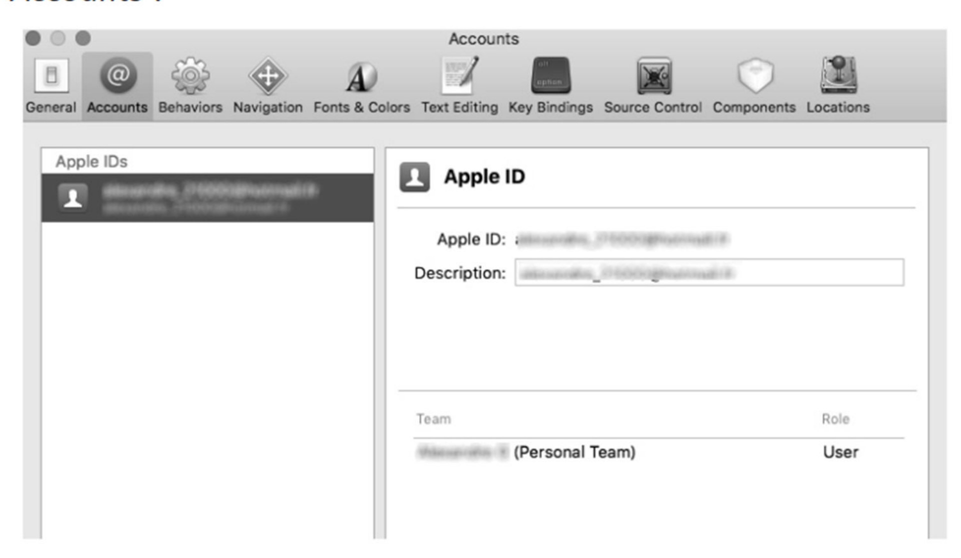


Si aucune erreur n'est affichée lors de l'appui sur « ok », c'est que la configuration est correcte.

Il faut encore éditer votre projet Cordova pour ajouter dans le fichier `build.json` des lignes spécifiques à iOS (par défaut il n'y a que des lignes pour Android). :

```
"ios": {                                                    {JSON}
     "debug": {
      "codeSignIdentity": "iPhone Developer",
      "developmentTeam": "XXXXXXXXXXXX",
      "packageType": "development",
      "buildFlag": [
        "EMBEDDED_CONTENT_CONTAINS_SWIFT = YES",
        "ALWAYS_EMBED_SWIFT_STANDARD_LIBRARIES=NO",
        "LD_RUNPATH_SEARCH_PATHS =
     \"@executable_path/Frameworks\"" ]
    }}
```

Pour la valeur de « `developmentTeam` » il faut inscrire le nom de votre équipe de développement associée avec votre compte développeur Apple. Pour la connaitre, lancez XCode sur le Mac, puis dans le menu *preferences*, choisir *Accounts* :



Capture 12 : compte développeur Apple sur XCode

L'émulateur se lance, hélas sur le Mac et non sur le PC, il faut donc que celui-ci soit accessible.

Et de plus, si l'exécution doit se faire sur un périphérique réel (iPhone, iPad, iPod) et non sur un émulateur, il faut encore quelques étapes… Tout ceci est certes très fastidieux, mais hélas obligatoire. Créez avec XCode sur le Mac un projet iOS avec le même nom de *package* que votre application, afin de créer les autorisations nécessaires de déploiement sur l'appareil. Dans Visual Studio, choisir « appareil distant » pour exécuter l'application sur un iPhone ou iPad connecté au Mac.

> Le premier lancement d'un simulateur iPhone pouvant être très long, il arrive qu'une erreur survienne dans Visual Studio (type simulateur introuvable) : il suffit de relancer le déploiement une fois le simulateur exécuté.

Il est également possible, si la connexion entre le PC et le Mac ne fonctionne pas correctement (ou si l'ensemble vous semble trop compliqué !) de faire générer l'application par Visual Studio, de copier sur le Mac le dossier `plateforms/ios` qui contient un projet XCode qui peut être exécuté directement.

e) Petite application (Cordova)

Cordova étant intégré à Visual Studio 2017, il est très pratique de l'employer avec cet éditeur. Néanmoins d'autres environnements sont utilisables sans difficulté.

Nous allons utiliser Visual Studio et Cordova pour écrire notre application « *hello world clic me* » avec les langages du web, HTML, CSS et Javascript.

Commencez par créer une application vide Cordova (Nouveau projet → Autres langages → Javascript → Mobile Apps → Application vide (apache cordova)).

L'assistant crée un projet contenant divers fichiers. Parmi ceux-ci, nous éditerons :

- Le fichier `www/index.html` qui contient la description de la page principale
- Le fichier `www/css/index.css` qui contient la feuille de style associée à la page principale
- Le fichier `www/scripts/index.js` qui contient le code Javascript associé à la page principale

Il faut tout d'abord éditer la page `index.html` pour décrire notre page : un texte, un bouton, un texte…

```
<body>
     <p>Hello, world</p>
    <button id="clic">Clic me</button>
    <p id="clics">0 clics</p>
    <script type="text/javascript" src="cordova.js"></script>
    <script type="text/javascript"
     src="scripts/platformOverrides.js"></script>
    <script type="text/javascript"
     src="scripts/index.js"></script>
</body>
```

L'aspect général devra être réglé dans la feuille de style. Nous ne nous en occuperons pas, mais vous pouvez modifier le fichier CSS pour avoir un aspect plus agréable.

Il faut ensuite associer au clic sur le bouton le code Javascript nécessaire au « comptage ». Pour cela, dans le fichier `index.js`, il faut ajouter les lignes suivantes à la fin de la fonction `onDeviceReady` :

```
var cpt = 0;
document.getElementById('clic').onclick = function() {
    ++cpt;
    document.getElementById("clics").innerHTML = cpt + " clic
     s";
};
```

Il est ensuite possible de tester l'application dans un navigateur, dans un simulateur ou dans un périphérique réel (l'application est certes peu attrayante, mais il suffirait de modifier le fichier CSS pour la rendre plus agréable à l'œil…).



Capture 13 : Hello World avec Cordova dans un navigateur, Windows 10, Android, iOS

4. Qt

Le *framework* Qt est fortement multiplateforme : il existe pour Windows (y compris les versions récentes, WinRT et UWP[1]), macOS, Linux, Android, iOS… l'environnement de développement intégré (EDI) Qt Creator existe pour Windows, Mac OS, Linux. Le langage utilisé est normalement C++ mais il est possible d'utiliser sous certaines conditions Java, Python ou d'autres langages encore.

Nous utiliserons ici C++ dans sa norme C++11.

Il existe une version *open source* des outils Qt qui peut donc être librement téléchargée et utilisée pour produire des applications. Je vous invite à consulter la licence d'utilisation, comme pour tout logiciel.

---

[1] *Universal Windows Platform* : plateforme Windows universelle. C'est le nouveau SDK de Microsoft pour le développement pour Windows 10, que ce soit sur PC, téléphone mobile, Xbox one, etc.

Qt est une des solutions les plus générales mais c'est également un des outils les plus complexes à utiliser. De plus, C++ n'est pas le langage le plus simple pour un débutant. C'est donc une solution déconseillée à un débutant ! Elle est cependant très intéressante pour tous ceux qui utilisent déjà Qt et/ou qui connaissent bien C++.

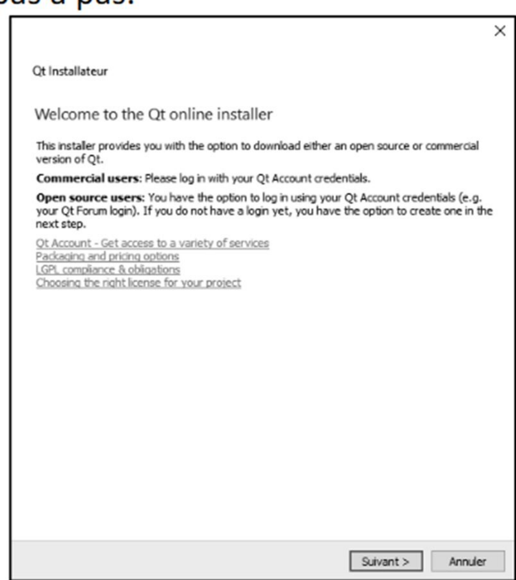
a) Installation de Qt

Quel que soit votre système d'exploitation, il faut déjà se rendre à cette adresse : https://www.qt.io/download-open-source. Le site devrait vous proposer la version la plus appropriée à votre système d'exploitation. La version *online* de l'installateur, qui télécharge uniquement les paquets désirés suivant vos choix, est à préférer. Les systèmes supportés sont :

- Windows 32 et 64 bit, versions 7 à 10 (développement possible pour Android, Windows, Windows 10 mobile, Windows phone)
- Mac OS X 64 bit, version plus récente que 10.6 (développement possible pour Android, Mac OS, iOS)
- Linux 64 bit (développement possible pour Android et Linux)

Il est important de noter que le code source est portable : si vous possédez un PC et un Mac, vous pouvez avec le même code faire une application pour Windows mobile, Android et iOS.

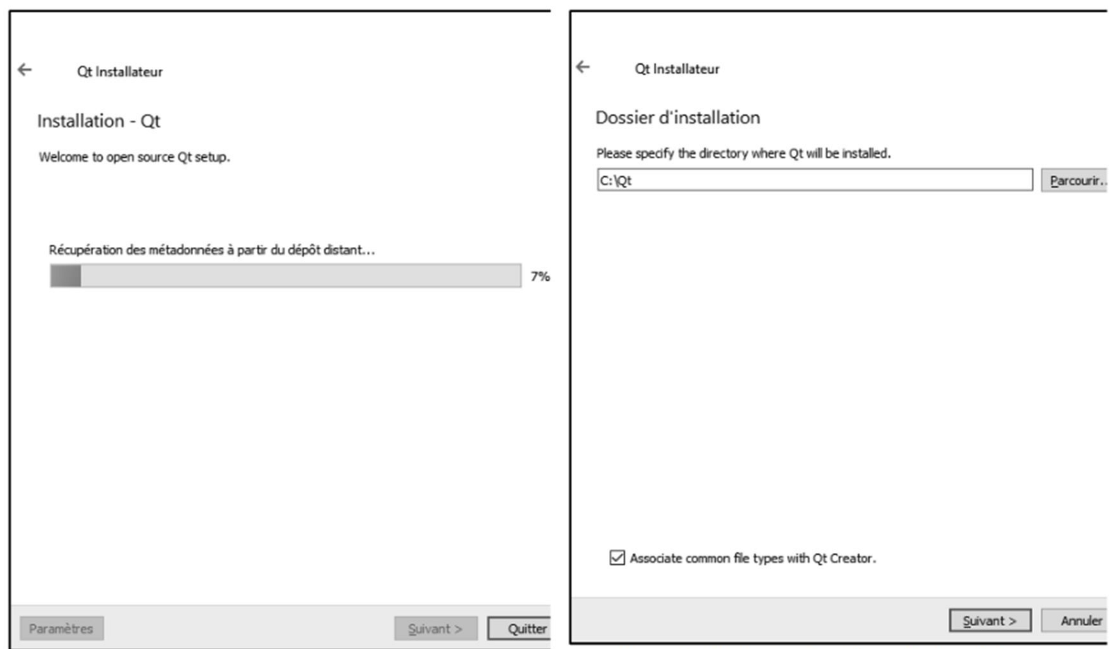
Une fois l'installateur téléchargé, il faut l'exécuter puis suivre les étapes pas à pas.



Il n'y a ici qu'à faire « suivant »

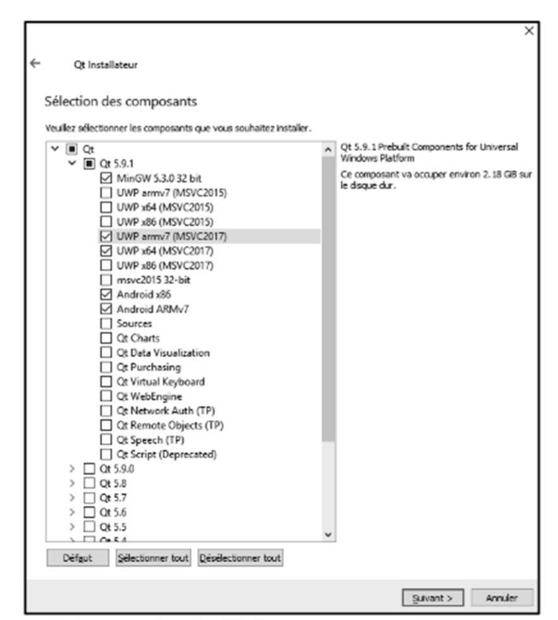


Vous pouvez créer un compte Qt mais ce n'est pas utile : cliquez « Skip »



Un peu de patience…

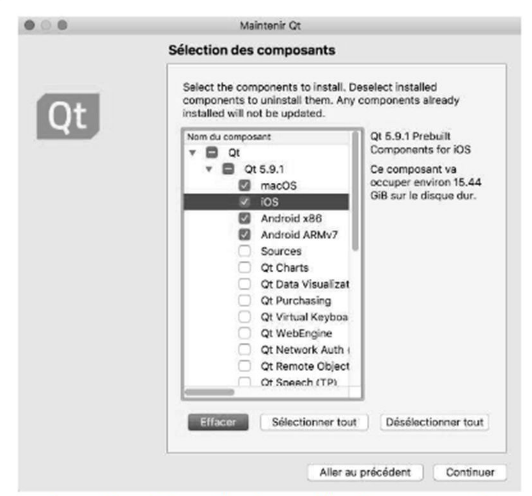
Il faut choisir le dossier d’installation (ici sous Windows). Evitez un dossier avec des droits restreints.



Il faut ensuite choisir les composants. Ici, sous un PC/Windows. MinGW pour faire une application « bureau », UWP arm pour un Windows 10 mobile, UWP x64 pour du Windows 10, Android x86 et arm pour de l’Android.



Sous Linux, seuls les outils pour les applications « bureau » (Desktop) et Android sont disponibles. La version x86 sera utile pour l’émulateur rapide et la version ARM pour un périphérique réel (tablette ou téléphone).



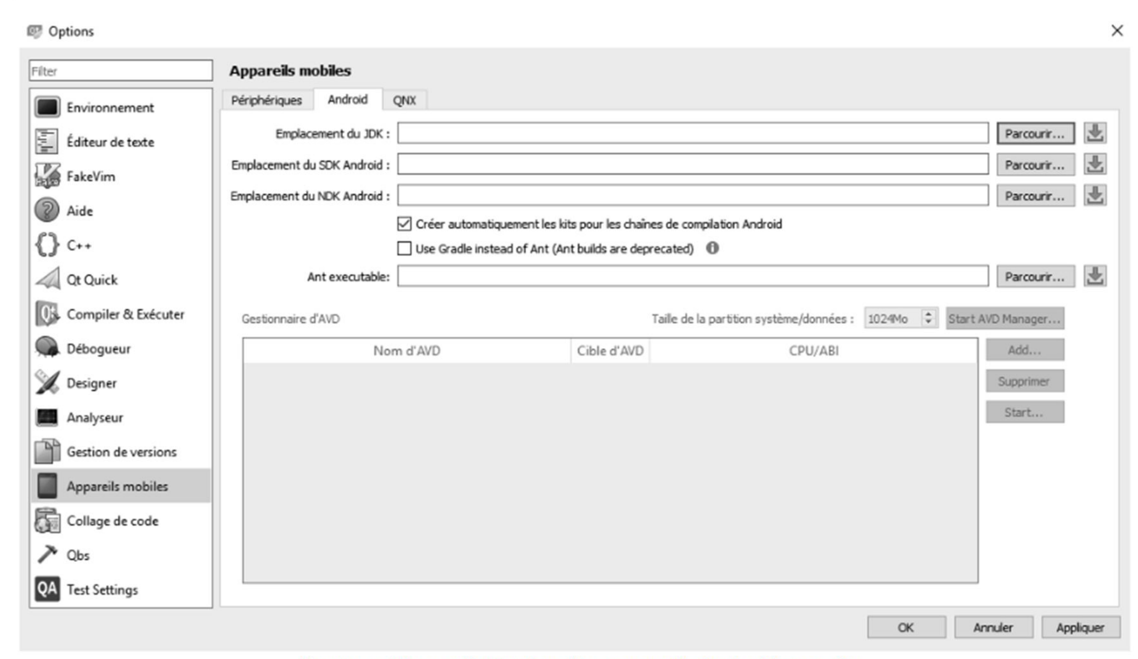
Sous Mac OS, seules les applications « macOS », « iOS » et « Android » sont possibles.



Après une (longue) attente, l'installation est terminée ! Lancez Qt Creator pour finir la configuration

Attention : l'installateur de Qt n'installe que le *framework* et l'EDI Qt Creator. Les kits de développements (Android, iOS, UWP) ne sont pas inclus et doivent être téléchargés par la suite si ce n'est pas déjà fait. Le plus simple pour installer un kit de développement est d'installer l'outil natif (Android Studio, XCode, Visual Studio).

Le reste de la configuration se fait dans Qt Creator qui devrait se lancer à la fin de l'installation. Il faut indiquer à Qt l'emplacement des divers kits de développement afin qu'il puisse réaliser correctement des applications Android, UWP, iOS (suivant votre système) ; ces emplacements sont en général automatiquement détectés mais il arrive que cela ne soit pas le cas et qu'il faille le faire manuellement. Pour cela, allez dans le menu « outils », puis « options → → appareils mobiles ». Pour Android, il faut indiquer les emplacements du JDK (le kit de développement Java), du SDK Android (kit de développement) ainsi que du NDK Android (kit de développement natif). Si vous avez déjà installé (avec un autre outil par exemple) ces différents outils, vous devez indiquer à l'aide du bouton « parcourir » leur emplacement, sinon un clic sur la flèche vous ouvrira un navigateur avec le site de téléchargement (patience : les téléchargements sont très lourds, plusieurs Go).

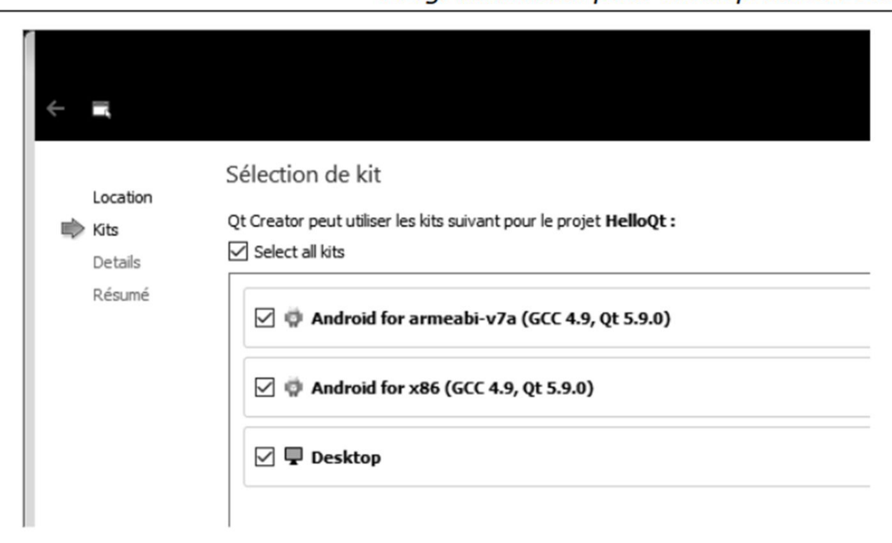


Capture 14 : configuration des appareils Android avec Qt

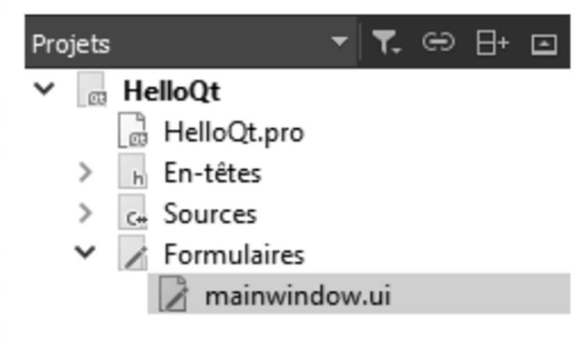
b) **Petite application**

Lorsque l'installation de Qt est terminée, nous pouvons créer notre petite application « *hello world, clic me* ».

Une fois Qt Creator exécuté, faire « nouveau projet », puis choisir le modèle de projet « application Qt avec widgets ». Choisir ensuite le dossier dans lequel les sources du projet seront placées, puis les kits choisis (les plateformes cibles). Si vous voulez une application Android, pensez bien à cocher la version x86 (pour l'émulateur) et la version ARM (pour le périphérique réel, téléphone ou tablette). Il est également possible de cocher la version « Desktop » ce qui permettra de tester le logiciel dans l'environnement normal (Windows, Linux ou macOS).



La fenêtre suivante permet de choisir le nom de classe principale de l'application Qt, celle qui représente l'écran (la fenêtre). Les noms par défaut (`MainWindow`) peuvent être laissés tels quels.

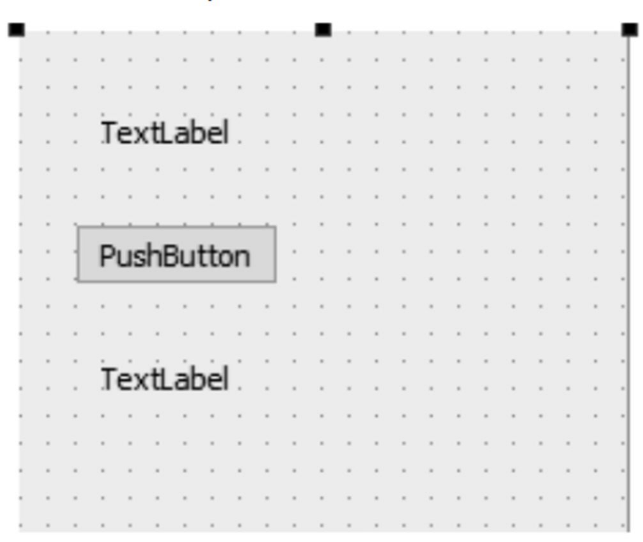


Une fois le projet créé, il faut éditer la partie visuelle de l'application, pour cela il faut double-cliquer, dans l'explorateur de projet, sur le dossier « formulaires » puis sur celui représentant la page (`MainWindow.ui` par défaut).

Qt Creator ouvre alors l'éditeur visuel d'interface. Nous allons créer pour un téléphone, donc plutôt pour un périphérique en mode « portrait » : redimensionnez la fenêtre proposée pour obtenir une largeur inférieure à la hauteur.

Il faut ensuite placer un certain nombre de contrôles sur la fenêtre. Dans notre cas, c'est très simple, il faut du texte pour dire « hello » (un `Label`), un bouton pour que l'utilisateur clique (`PushButton`) et du texte pour donner le nombre de clics. Glissez-déposez sur la fenêtre les composants nécessaires, sans vous occuper de leur placement pour l'instant.

Le résultat ressemble à la capture ci-dessous :



Capture 15 : Design "hello world" Qt

Nous allons à présent indiquer que le placement des contrôles doit suivre un « flot » vertical (ils seront alignés les uns au-dessous des autres). Pour cela, faire un clic droit sur la fenêtre, choisir dans le menu contextuel « Mettre en page » puis « Mettre en page verticalement ». Les composants se positionnent alors automatiquement.

Il faut éditer ensuite les paramètres (propriétés) de chaque contrôle : sélectionnez un contrôle à la souris, puis éditez les paramètres voulus dans la zone en bas à droite. Vous pouvez changer le nom de l'objet (nom de la variable dans le code) et un grand nombre de propriétés.

Commençons par la fenêtre elle-même : les propriétés à modifier sont :

- `WindowTitle` (le titre de la fenêtre) : par exemple « `Hello Qt` »
- `Font` (la police utilisée pour la fenêtre) : par exemple « `Segoe UI, 20 pts` »



Pour l'objet `Label` du haut, réglez les propriétés suivantes :

- `text : « Hello world »`
- `objectName : « hello »`

Pour le bouton :

- `text : « clic me »`
- `objectName : « clic »`

Pour l'objet `Label` du bas :

- `objectName : « clics »`
- `text : « 0 clic »`

L'aspect général de la fenêtre doit ressembler à l'image ci-dessus.

Vous pouvez exécuter l'application (le plus rapide : en mode *Desktop*) pour voir l'effet à l'écran. Il faut ensuite répondre au clic de l'utilisateur sur le bouton. Pour cela, dans l'éditeur visuel, faire un clic droit sur le bouton, choisir « aller au *slot* » puis « `clicked()` ». L'éditeur va créer une fonction C++ (intitulée `on_clic_clicked`) qui sera appelée à chaque clic de l'utilisateur sur le bouton. Dans cette fonction, saisissez le code C++ suivant :

```
static int cpt = 0;
++cpt;
QString msg = QString::number(cpt) + " clics";
ui->clics->setText(msg);
```

La première ligne déclare une variable statique[1] initialisée à 0, qui va servir de compteur. Ce compteur est ensuite incrémenté à l'aide de l'opérateur ++.

[1] Une variable statique en C++ est persistante entre deux appels de la fonction.

La 3ème ligne déclare une variable de type `QString`[1] qui est le résultat de la concaténation de la chaîne produire par le compteur (utilisation de `QString::number` pour convertir l'entier en chaîne) et le texte « `clics` ».

La dernière ligne permet de modifier la propriété `text` du contrôle `clics`.

Vous pouvez exécuter l'application (en mode *Desktop*) pour tester le résultat. Si tout marche, vous pouvez essayer ensuite d'exécuter l'application dans un émulateur ou dans un vrai téléphone. Pour l'émulateur Android, par exemple, choisir la cible « Android for x86 ». A l'exécution une fenêtre vous proposera de choisir l'émulateur (si un émulateur a été créé) ou le périphérique (si le périphérique est branché) à utiliser ; vous pourrez également créer un nouvel émulateur.

Le résultat, dans un émulateur, est le suivant :





Capture 16 : "hello world" avec Qt pour Android et iOS

> Armez-vous de patience : la compilation pour un appareil Android ou iOS peut être très longue. Si la cible est un émulateur, il faut aussi le temps d'exécuter celui-ci.

---

[1] QString est le type chaîne de caractères de Qt.

c) Android : spécificités

L'édition du manifeste Android (qui contient, entre autres, la définition des visuels, les autorisations, etc. voir XI.1.a)) peut se faire depuis Qt Creator. Il faut ouvrir les options de configuration (Barre d'outils → Projets) pour la configuration Android (x86 et/ou arm), dérouler l'option « *build android apk* », puis « *create templates* ». Qt Creator vous propose alors de créer un dossier avec les sources android nécessaires (par défaut, dans le sous-dossier `android` de votre projet). Vous pouvez ensuite éditer le manifeste directement, pour, entre autres :

- Choisir le nom du paquet
- Choisir la version minimale de l'API
- Choisir la version cible de l'API
- Régler les autorisations de l'application (la plupart sont incluses automatiquement par Qt)
- Choisir les visuels (icône par exemple)

d) Windows 10 : spécificités

Le manifeste Windows (voir XI.1.c) n'est pas éditable en tant que tel, mais au travers d'options dans le fichier projet .PRO. Les options les plus importantes (5) sont les suivantes :

- `publisher` : le nom de l'éditeur / développeur
- `publisher_id` : l'identifiant de l'éditeur (donné par le MS Store)
- `identity` : ID unique de l'application (donné par le MS Store)
- `background` : couleur de fond de l'arrière-plan (vert par défaut)
- `logo_store` : le fichier image (PNG) pour le logo du MS Store
- `logo_splash` : le fichier image (PNG, 620x300) pour l'écran de présentation
- `logo_large` : l'icône de l'application (PNG, 150x150)
- `logo_medium` : idem mais en 70x70
- `logo_small` : idem mais en 30x30

Toutes les options doivent être préfixées par `WINRT_MANIFEST`, par exemple :

```
WINRT_MANIFEST.logo_store = icon.png
```

e) iOS : spécificités

Lors de la compilation, si la cible est un périphérique (iPhone ou iPad), alors Qt ne pourra ni la compiler ni la déployer et indiquera une erreur du type « *Xcode unable to find provisionning profile for xx.yy.zz* ». Cela signifie que le compilateur d'Xcode (utilisé par Qt pour produire l'application) ne trouve pas le profil pour l'application. Il faut le créer avec XCode : il suffit pour cela de créer une application vide en lui donnant le même nom complet (par exemple `com.ag.hello-qt`) que dans l'application Qt (voir le message d'erreur précédent).

Attention : la dernière version de Qt (5.11 à l'heure où ces lignes sont écrites) ne supporte iOS qu'à partir de la version 10. Le tableau suivant indique les versions de iOS supportées par Qt (6) :

Version de Qt	Versions de iOS supportées
5.11	10
5.10	10
5.9	8, 9, 10
5.8	7
5.7	6
5.6	6
5.5	5

Suivant l'appareil ciblé (et donc la version d'iOS) il peut être nécessaire de choisir un kit plus ancien.

Qt ne proposant pas d'éditeur pour le manifeste iOS (voir XI.1.b), il faut modifier directement celui-ci : c'est le fichier `Info.plist`. Ce fichier est automatiquement généré par Qt mais il est possible, s'il est besoin de le modifier, d'indiquer un fichier personnel dans le fichier projet (.pro) de Qt :

```
ios {
QMAKE_INFO_PLIST = ios/Info.plist
}
```

Plutôt que de créer ce fichier, il est plus simple de récupérer celui généré par Qt à la première compilation de votre projet (il faut donc compiler…) il est placé dans le dossier de construction (généralement un dossier dont le nom commence par `build` et qui se trouve au même niveau que votre dossier source). Vous pourrez ensuite le modifier au besoin.

V. Programmation des IHM

Les IHM (interactions homme-machine) sont un point fondamental des applications mobiles. C'est la partie de l'application que voit et « touche » l'utilisateur, c'est par elle qu'il peut agir.

L'IHM possède deux fonctionnalités fondamentales :

- Informer l'utilisateur de l'état de l'application
- Informer l'application des souhaits de l'utilisateur

Une IHM est composée d'un certain nombre de **contrôles** qui sont des entités d'interactions, des éléments visuels apparaissant sur l'écran du téléphone et permettant à l'utilisateur de contrôler l'application.

Suivant le système, il existe un grand nombre de contrôles qui peuvent avoir des aspects très différents et des comportements divers ; néanmoins, dans les grandes lignes, nous utilisons toujours peu ou prou les mêmes types de contrôles et il est possible de généraliser la plupart du temps.

Dans cette partie, nous verrons les fondements des IHM sur téléphone, les notions de programmation évènementielle, les composants les plus courants et comment programmer le passage d'un écran à l'autre.

1. Notion d'écran

Sur téléphone comme sur ordinateur, une application interagit avec l'utilisateur. Sur ordinateur, avec une fenêtre ; sur le web on parle de page, et sur téléphone, d'écran. Une application est donc, du point de vue de l'utilisateur, composée d'un ou plusieurs écrans.

Chaque système utilise son propre vocabulaire et ses propres notions pour désigner les écrans. De plus, chaque *framework* utilise une classe pour l'écran avec un comportement et une manière de programmer différents… Néanmoins il existe de nombreux points communs :

- L'application contient au moins un écran
- L'utilisateur peut passer d'un écran à l'autre, revenir à l'écran précédent
- Les différents contrôles se positionnent sur un écran
- L'écran est associé à une classe qui contient le code lié à l'écran

a) Android : les activités

Sous Android les écrans sont appelés des **activités**. Chaque activité est une part autonome de l'application. Les activités sont des objets dont la classe hérite plus au moins directement de la classe `Activity`. Une activité contient une vue (ce que voit l'utilisateur) qui peut soit être construite par code, soit être décrite dans un fichier XML (ou visuellement par glisser-déposer).

Quand on crée une activité dans Android Studio, un fichier `.java` et un fichier `.xml` sont créés. Le fichier XML peut être édité « à la main » ou avec l'éditeur visuel de l'EDI.

La vie de l'activité (création, exécution, mise en pause, arrêt, destruction) est gérée par le système : dans certains cas, l'application peut être fermée sans que l'utilisateur l'ait demandé et c'est au programme de faire les sauvegardes qui s'imposent.

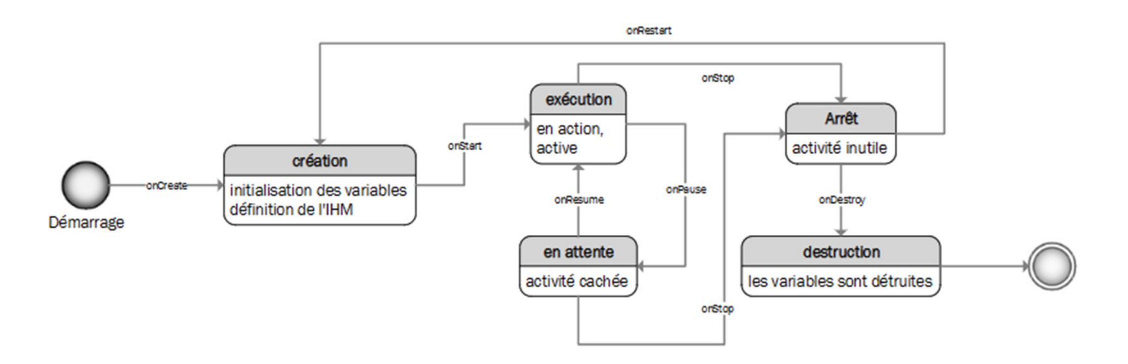


Diagramme 8 : cycle de vie d'une activité Android

A chaque fois que l'activité change d'état, une opération de sa classe est appelée :

- `onCreate` : création de l'activité. Opération obligatoire. Initialise l'activité (ne pas utiliser le constructeur pour cela).
- `onStart` ou `onRestart` : démarrage ou redémarrage de l'activité. L'état doit être récupéré d'une sauvegarde.
- `onPause` : l'activité est mise en pause (en général parce qu'elle passe à l'arrière-plan). Il faut suspendre les opérations non utiles pour économiser la batterie.

- `onResume` : l'activité sort de la pause (en général parce qu'elle passe au premier plan). Les opérations suspendues par la pause reprennent.
- `onStop` : l'activité est arrêtée. L'état doit être sauvegardé s'il doit être repris dans un redémarrage.
- `onDestroy` : l'activité va être détruire et retirée de la mémoire.

Dans chaque application Android, il est nécessaire de redéfinir `onCreate` dans chaque activité. Typiquement dans cette opération, il y aura :

- L'appel du `onCreate` de l'ancêtre : obligatoire pour initialiser l'activité,
- La définition de la vue (`ContentView`), en général par un fichier `.xml` de type `layout`,
- L'initialisation des attributs de la classe, notamment ceux qui représentent les contrôles.

Le lien entre un contrôle défini dans le *layout* (le fichier XML) et la classe java de l'activité se fait par un **identifiant** qui est unique pour l'activité. Chaque contrôle Android possède en effet une propriété `android:id` qui code son identifiant. Les identifiants déclarés dans le fichier XML du *layout* sont accessibles dans le code Java via une classe auto-générée appelée `R` (comme *Resources*) et une propriété `id` listant les identifiants.

Prenons par exemple une zone de saisie de texte (type `TextEdit`) définie comme suit dans le fichier `activity_main.xml` :

```
<TextEdit android:id="@+id/nom" />
```

Dans le fichier java de l'activité, il sera nécessaire de déclarer un attribut de type `TextEdit` pour manipuler le contrôle (l'attribut est ici nommé avec le même texte que le contrôle mais ce n'est pas obligatoire) :

```
private TextEdit nom ;
```

Dans l'opération `onCreate` de l'activité, après avoir appelé le `onCreate` de l'ancêtre (la superclasse) et initialisé la vue, il faudra initialiser l'attribut susnommé en utilisant l'identifiant :

```
public void onCreate(Bundle saved)
{
     super.onCreate(saved) ;
     setContentView(R.layout.activity_main) ;
     nom = (TextEdit)findViewById(R.id.nom) ;
}
```

b) Windows : les pages

Sous Windows 10, les différents écrans sont représentés par des classes qui héritent plus ou moins directement de la classe `Page`. Chaque page utilise un fichier pour décrire sa vue (type XAML) et un fichier contenant le code associé (type CS).

Les pages n'ont pas de cycle de vie autonome, c'est l'application qui en a. Son cycle est le suivant :

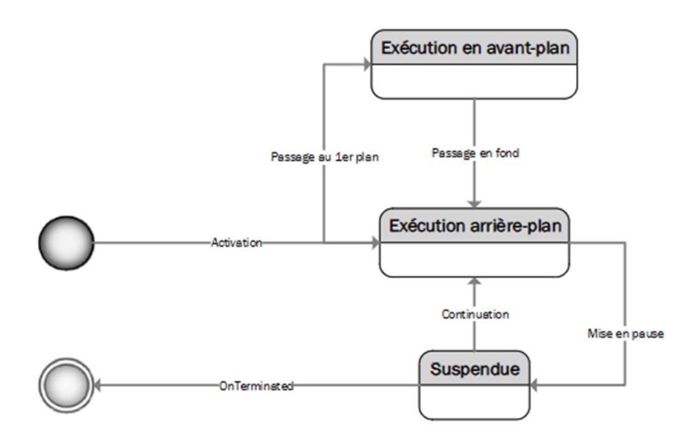


Diagramme 9 : Cycle de vie d'une application UWP

Pour chaque changement d'état, la classe `Application` définit un évènement qu'il est possible d'utiliser :

- `OnLaunched` : l'application est lancée et initialisée.
- `OnSuspending` : l'application est mise en pause. Elle peut être déchargée de la mémoire à tout instant, il faut donc enregistrer son état et arrêter toute activité (*thread*…)
- `OnActivated` : l'application est activée ou passe au 1er plan. Il faut redémarrer les activités de premier plan.
- `OnBackgroundActivated` : l'application passe à l'arrière-plan. Il faut stopper les activités de premier plan pour ne conserver que des activités d'arrière-plan.

La classe `Page`, quant à elle, possède deux opérations qu'il est possible de redéfinir :

- `OnNavigatedFrom` : l'utilisateur demande à changer de page, celle-ci va donc être désactivée
- `OnNavigatedTo` : l'utilisateur navigue depuis une autre page

Le lien entre les contrôles déclarés dans le `.xaml` et le code `.cs` de la page se fait très simplement grâce à la propriété `x:Name` du contrôle : un attribut du type *ad hoc* est automatiquement créé dans la classe.

Prenons le cas d'une zone de saisie de texte (type `TextBox`) déclarée comme suit dans le fichier `.xaml` :

```
<TextBox x:Name="nom" />
```

Dans le code de la classe, il sera très simple d'accéder à ce contrôle par l'attribut `nom`, sans avoir eu besoin de le déclarer :

```
String s = nom.Text ;
```

Les initialisations de la page se font dans le constructeur. Celui-ci contient obligatoirement l'appel à l'opération `InitializeComponent` qui créée les contrôles à partir du fichier `xaml` associé.

c) iOS : les vues et le *storyboard*

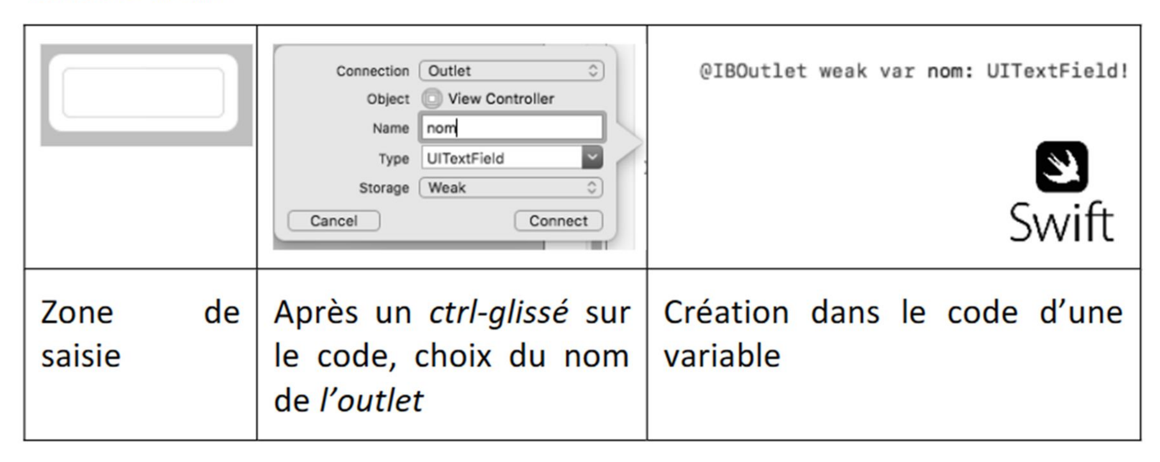
Le vocabulaire utilisé par *XCode* est assez différent des autres outils. Pour représenter les écrans, on emploie les termes de vues (un écran), *storyboard* (cheminement entre les écrans) et de contrôleur (le code associé à une vue).

La vue se « dessine » à l'aide de l'outil graphique de *XCode*. Le fichier produit est en fait un fichier XML.

Une application « *single view* » (simple vue) ne contient donc qu'une seule vue intégralement définie dans le fichier `main.storyboard`. Le code associé est quant à lui placé dans le fichier `ViewController.swift`.

Le lien entre un contrôle de la vue et le code du contrôleur se fait en glissant-déplaçant le contrôle dans le code, ce qui créée un lien (type `Outlet`) entre les deux.

Si nous avons, par exemple, une zone de saisie (type `UITextField`) appelée `nom` dans la vue, en la glissant dans le code du contrôleur et en créant un `outlet`, il sera ensuite possible d'y accéder depuis le code par la variable `nom`.



Zone de saisie	Après un *ctrl-glissé* sur le code, choix du nom de *l'outlet*	Création dans le code d'une variable

d) Xamarin.Forms : les pages

Le système de *Xamarin.Forms* est quasiment identique au système UWP vu au V.1.b). Les types ne sont pas exactement les mêmes, mais le principe est similaire.

e) Qt : les fenêtres

Qt possède deux manières de gérer l'interface graphique : par des *widgets* ou à l'aide du langage QML. Nous utiliserons ici les *widgets*, plus simples et (dans la plupart des cas) plus pratiques.

Étant donné son origine « bureau », Qt parle de fenêtres et non d'écran ou de page, mais les fenêtres sont traduites en activités sous Android, en pages sous Windows, etc… Elles sont en plein écran sur les terminaux mobiles.

Qt utilise la classe `QWindow` pour les fenêtres. L'initialisation se fait dans le constructeur, et la libération mémoire (souvenons-nous que C++ n'a pas de *garbage collector*[1]) dans le destructeur.

---

[1] *Garbage collector* : en français ramasse-miettes ; partie du système qui, en arrière-plan, traque les références mémoires inusitées pour libérer la mémoire. Existe en Java et en C# notamment.

Pour chaque fenêtre, Qt va créer une classe héritant de `QWindow` (plus ou moins directement) dans un couple de fichiers (`.h`/`.cpp`) ainsi qu'un formulaire dans un fichier `.ui` (format XML, mais un éditeur visuel est fourni).

Le lien entre le contrôle défini dans le formulaire et la classe se fait tout simplement par la propriété `name` de chaque contrôle : un attribut du type correct est automatiquement défini dans une classe interne à la fenêtre.

Si nous avons, par exemple, une zone de saisie (type `QLineEdit`) appelée `nom`, nous y accédons dans la classe de la fenêtre via l'attribut `nom` de la propriété `ui` :

```
QString n = this->ui->nom->text() ;
```

f) Cordova : les pages

Cordova utilisant HTML, la notion d'écran est identique à celle d'un site web: on parle de page. Pas de classe pour représenter cette notion ici, chaque fichier `.html` représente une page différente (il est également possible de générer la page uniquement par *Javascript*). La page (le fichier HTML) est associée à une ou plusieurs feuilles de style (fichiers CSS) ainsi qu'à un ou plusieurs fichiers de script (fichiers JS ou TS) qui contiennent du code exécutable.

Les contrôles définis en HTML sont accessibles par le code *Javascript* soit en utilisant leur **identifiant** (unique pour chaque contrôle), soit en utilisant un sélecteur CSS, ce qui permet d'en choisir plusieurs à la fois. Cette dernière possibilité est un peu particulière et ne sera pas décrite ici. Nous utiliserons donc l'accès via l'identifiant.

Exemple d'une zone de saisie texte identifiée par « `nom` » et définie dans le fichier `.html` :

```
<input type="text" id="nom" />
```

Dans un fichier .JS lié à ce fichier HTML, l'objet *javascript* associé au contrôle sera accessible grâce à l'opération `getElementById` de l'objet document :

```
var nom = document.getElementById("nom") ;
```

Il est possible d'utiliser la bibliothèque *JQuery* pour simplifier cette écriture, mais ce n'est pas le propos de ce livre.

2. Contrôles et composants

L'utilisateur interagit avec l'application au travers d'un certain nombre de **contrôles** qui composent l'interface utilisateur (ou interface homme-machine). Ces contrôles sont de natures diverses et dépendent fortement de la plateforme et/ou du *framework* utilisé. Chaque API ou *framework* propose un certain nombre de **composants** logiciels créés pour manipuler efficacement les contrôles.

Il y a des contrôles très classiques, que l'on retrouve pratiquement dans toutes les plateformes et tous les *frameworks*. Nous allons rapidement étudier les plus courants.

Notez que l'aspect de chaque contrôle dépend principalement de la plateforme (du système, de sa version) mais aussi de l'outil utilisé. En effet, certains *frameworks* utilisent les contrôles natifs (du système) mais d'autres recréent les leurs. Il est également possible de paramétrer l'affichage des contrôles au moyen de thèmes (ou de feuilles de styles avec Cordova et Qt, par exemple).

L'aspect général des contrôles peut également être très variable au sein de la même plateforme : les propriétés de ces contrôles peuvent être modifiées, certains thèmes (sombres ou clairs, par exemple) permettent de changer l'aspect complet de l'application. Le tableau de la page suivante donne donc un aperçu des contrôles standards de chaque plateforme, avec les valeurs par défaut de leurs propriétés.

a) Le texte simple (étiquette, *label*…)

Le premier d'entre eux est également le plus simple : l'étiquette (ou *label*) est un texte simple qui ne présente donc que peu d'interactions avec l'utilisateur. Son but est de fournir une information de type textuel. Ses propriétés classiques sont bien entendu le texte affiché, la police de caractères utilisée, son alignement.

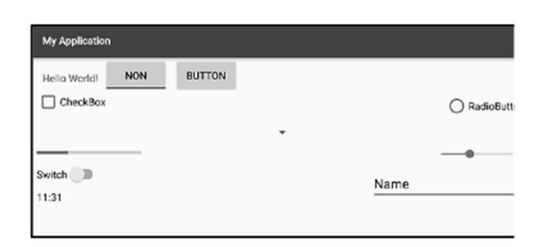
Le tableau fournit les différents composants associés à ce contrôle pour chaque outil utilisé :

Tableau 5 : le contrôle Label avec chaque outil

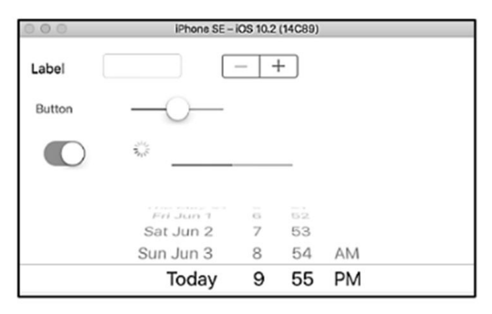
UWP	Android	iOS	Xamarin	Cordova	Qt
`TextBlock`	`TextView`	`UILabel`	`Label`	`label, p`	`QLabel`



Windows universal platform



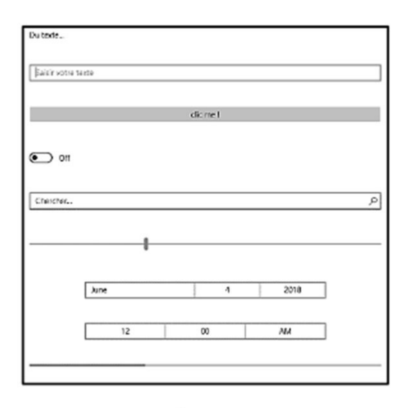
Android



iOS



Qt Widgets - Windows



Xamarin.Forms



HTML5

Tableau 6 : aspect des contrôles standard sous chaque plateforme

b) La zone de saisie

Il est extrêmement fréquent de devoir faire saisir à l'utilisateur du texte, un nombre, une valeur quelconque, par le biais d'un clavier (physique ou virtuel). Un contrôle de type zone de saisie existe sur chaque plateforme. Ses propriétés classiques sont le texte saisi (ou affiché), la police de caractères utilisée, son alignement, son type de saisie (numérique, texte, mail, etc.). Il peut être sur une seule ligne ou sur plusieurs lignes.

Le tableau fournit les différents composants associés à ce contrôle pour chaque utilisé :

Tableau 7 : le contrôle de saisie sur chaque outil

<table>
<tr><th>Lignes</th><th>UWP</th><th>Android</th><th>iOS</th><th>Xamarin</th><th>Cordova</th><th>Qt</th></tr>
<tr><td>1</td><td rowspan="2">TextBox</td><td rowspan="2">EditView</td><td rowspan="2">UITextField</td><td>Entry</td><td>input</td><td>QLineEdit</td></tr>
<tr><td>+</td><td>Editor</td><td>textarea</td><td>QTextEdit</td></tr>
</table>

c) La case à cocher

Ce type de contrôle permet de saisir de manière très simple une valeur booléenne (vrai/faux, oui/non…). Il se présente classiquement sous la forme d'une case cochée ou non mais dans les IHM mobiles, c'est plus souvent une sorte d'interrupteur à glissière. Le rôle est néanmoins identique. Sa principale propriété est bien entendu son état « vrai » ou « faux ».

Le tableau fournit les différents composants associés à ce contrôle pour chaque outil utilisé :

Tableau 8 : contrôle case à cocher sur chaque outil

UWP	Android	iOS	Xamarin	Cordova	Qt
CheckBox	CheckBox	UISwitch	Switch	input	QCheckBox

d) La liste de choix

Il est très courant de représenter une liste de valeurs, notamment pour en faire choisir une à l'utilisateur. La liste peut s'afficher sous différentes formes, suivant les systèmes, mais son principe est similaire. Elle contient

soit des chaînes, soit des objets (et dans ce cas, les objets doivent pouvoir se convertir en chaînes). Certains *frameworks* poussent la complexité et proposent de paramétrer l'affichage de chaque *item* de la liste. Les listes peuvent aussi être montrées partiellement (déroulantes, coulissantes, surgissantes…). Les propriétés principales sont bien entendu la liste des *items* ainsi que la référence (position, adresse, etc…) de celui choisi (sélectionné) par l'utilisateur.

Le tableau fournit les différents composants associés à ce contrôle pour chaque outil utilisé :

Tableau 9 : le contrôle liste sur chaque outil

UWP	Android	iOS	Xamarin	Cordova	Qt
`ListBox`	`Spinner`	`UIPicker`	`ListView`	`select, option`	`QListWidget, QComboBox`

e) Le bouton

Moyen le plus simple pour l'utilisateur d'agir sur l'application, le bouton est un contrôle qui permet de déclencher une action. Il peut afficher un texte et/ou une image, être associé à un évènement, peut avoir deux états ou non, etc.

Le tableau fournit les différents composants associés à ce contrôle pour chaque outil utilisé :

Tableau 10 : le contrôle bouton sur chaque outil

UWP	Android	iOS	Xamarin	Cordova	Qt
`Button`	`Button`	`UIButton`	`Button`	`button, input`	`QPushButton, QToolButton`

f) Et les autres contrôles ?

Inutile (impossible ?) de donner une liste exhaustive ici… Dans chaque outil, avec chaque *framework* et sur chaque plateforme, vous trouverez une liste plus ou moins complète des contrôles disponibles et de leurs propriétés. N'oubliez pas de consulter la documentation de votre outil !

3. Notion de disposition d'écran

L'écran (ou la page, la fenêtre…) est le premier contrôle que l'on rencontre dans une application, et le seul obligatoire. Une application est obligatoirement composée d'un écran au minimum (mais il peut en avoir plusieurs).

Un écran est un conteneur de contrôles : c'est lui qui va recevoir le placement de ceux-ci. Il faut donc bien comprendre comment ceux-ci vont se positionner.

Il existe plusieurs manières de disposer des contrôles sur un écran. Ces méthodes peuvent être utilisées ensemble, sur différentes parties du même écran.

Dans chaque outil de développement, la méthode utilisée pour choisir une disposition diffère à la fois dans la forme et dans le fond.

Le choix de la meilleure disposition et de ses paramètres ne rentre pas dans le cadre de cet ouvrage, de même que les principes d'ergonomie et/ou d'esthétisme.

a) Disposition absolue

Dans ce mode, chaque contrôle est placé à des coordonnées précises de l'écran, en général exprimées en pixels[1] par rapport au coin en haut à droite de l'écran. Ce mode est le plus « direct » mais le moins pratique sur un téléphone, à cause des nombreuses tailles d'écran possibles.

b) Disposition linéaire

Dans ce mode, les contrôles sont disposés à la suite les uns des autres, suivant une ligne horizontale ou verticale.

Ce mode est approprié pour une liste homogène (3 boutons par exemple).

---

[1] Pixel : contraction de *Picture Element*, élément d'image. Point de base d'une image sur un écran, plus petit élément d'image possible.

c) Disposition relative

Dans ce mode, les contrôles sont disposés les uns par rapport aux autres. Le premier contrôle posé servant de référence, les suivants se placent par rapport à lui. On indiquera, par exemple, qu'un bouton est placé « après » une image ; suivant la dimension de l'écran, le bouton sera à droite ou en-dessous de l'image.

Ce mode est approprié pour une page adaptative supportant les changements d'orientation mais ne garantit pas un aspect similaire.

d) Disposition en bordure

Dans ce mode, les contrôles sont disposés par rapport aux bords de l'écran. Tout se passe comme si les bords étaient « aimantés » ; pour chaque contrôle on indique le bord de référence (bas, haut, gauche, droite, ou le « milieu ») et le contrôle se place sur lui, en adoptant la dimension *ad hoc*.

Ce mode est approprié pour la disposition générale d'un écran, surtout sur un ordinateur, moins sur un téléphone.

e) Disposition en grille

Dans ce mode, l'écran est découpé suivant une grille et chaque contrôle se positionne dans une cellule (une case) de cette grille. Ce mode est le plus général et le plus approprié à une IHM avec un aspect le plus constant possible.

f) Les dispositions sous Android

Les vues Android utilisent la classe `Layout` pour configurer les dispositions. Chaque contrôle va posséder des paramètres qui règlent la manière dont il se place au sein de son conteneur.

La largeur du contrôle se règle avec `layout_width` et peut prendre les valeurs :

- `wrap_content` : prend l'espace minimal nécessaire pour son contenu
- `match_parent` : prend l'espace maximal disponible dans son parent

- Une valeur précise exprimée en `dp`[1]

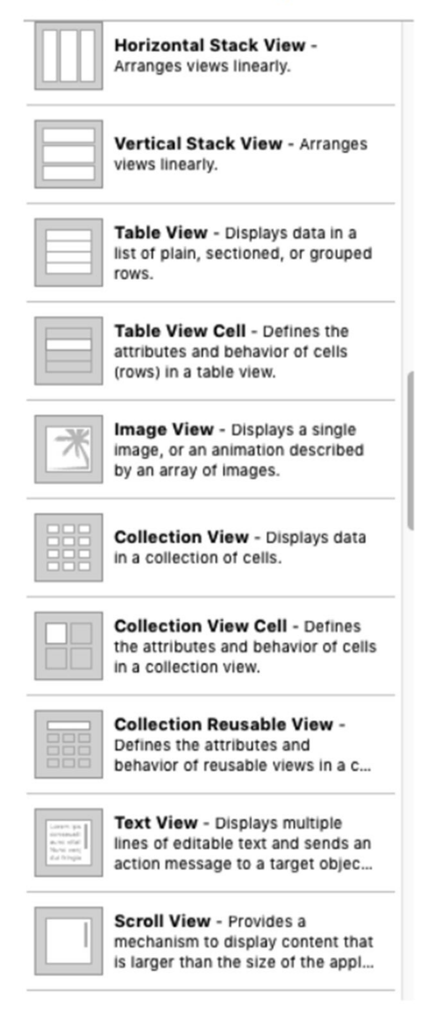
La hauteur du contrôle, réglée avec `layout_height` répond aux mêmes principes.

Il est également possible de préciser comment un contrôle se place relativement à son parent avec le paramètre `layout_gravity` qui peut prendre les valeurs suivantes (elles peuvent être combinées entre elles avec l'opérateur `|`) :

- `left` : le contrôle se place le plus à gauche possible
- `center_horizontal` : le contrôle se place horizontalement au centre du parent
- `right` : le contrôle se place le plus à droite possible
- `top` : le contrôle se place le plus en haut possible
- `center_vertical` : le contrôle se place verticalement au centre du parent
- `start` : le contrôle se place au début (équivalent à `left` ou `top` suivant le cas)
- `end` : le contrôle se place à la fin (équivalent à `right` ou `bottom` suivant le cas)

[1] *Density-independant pixel* : unité logique représentant un pixel indépendamment de la densité d'affichage, afin que les dimensions soient les mêmes quelque que soit le périphérique.

g) Les dispositions sous iOS



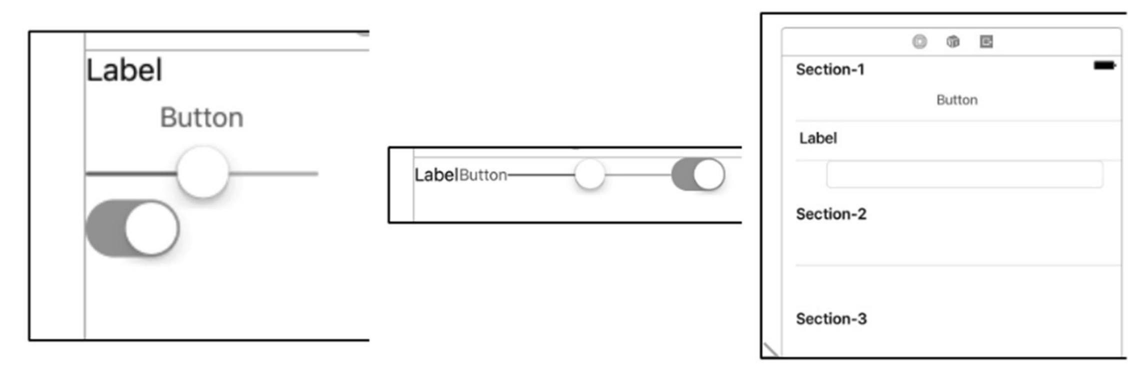
Capture 17 : dispositions sous XCode

iOS possède différentes possibilités pour disposer les contrôles. Par défaut, la disposition est automatique : les composants sont placés « à la souris » et des contraintes sont automatiquement créées. Cela marche bien dans la plupart des cas… mais pas pour tous.

Différentes dispositions existent, et il suffit de glisser-déposer l'une d'entre elles dans la vue. Cette disposition peut ensuite accueillir différents contrôles. La capture ci-contre indique les dispositions utilisables :

- *Horizontal Stack* : linéaire horizontale
- *Vertical Stack* : linéaire verticale
- *Table* : suivant différentes lignes
- *Collection* : suivant différentes lignes et colonnes
- *Scroll* : avec défilement (*scroll*).

La plupart du temps, l'*auto-layout* est suffisant. Pour certains écrans (ou partie d'écran) il peut être intéressant d'utiliser une disposition (linéaire par exemple).



Capture 18 : dispositions verticale, horizontale et table sous XCode

h) Les dispositions avec Xamarin.Forms

Xamarin.Forms fournit un certain nombre de **conteneurs** pour gérer différents types de disposition.

Tous les conteneurs héritent de la classe `Layout` (7). Nous trouvons les différents types de dispositions vus plus haut :

- `ScrollView` est un conteneur défilable qui peut donc contenir plus de contrôles que l'écran lui-même ; il définit un sous-*layout*
- `AbsoluteLayout` permet un placement absolu des contrôles
- `Grid` permet un placement en grille
- `RelativeLayout` permet un placement relatif
- `StackLayout` permet un placement en ligne (horizontale ou verticale)

StackLayout

Le placement en ligne peut s'effectuer suivant deux orientations : verticale et horizontale ; ce réglage est défini par le paramètre `Orientation` du `StackLayout`.

Les contrôles à l'intérieur du `StackLayout` doivent décider comment ils se placent de manière horizontale et verticale. Les marges peuvent également être définies.

Un contrôle peut être, horizontalement et verticalement :

- Centré : `Center`
- Centré en occupant le plus d'espace possible : `CenterAndExpand`
- Placé au départ : `Start`
- Placé au départ en occupant le plus d'espace possible : `StartAndExpand`
- Placé à la fin : `End`
- Placé à la fin en occupant le plus d'espace possible : `EndAndExpand`
- Justifié sur tout l'espace : `Fill`

```
<StackLayout Orientation="Vertical">
 <Label Text="Titre !"
 VerticalOptions="Start"
     HorizontalOptions="CenterAndExpan
     d"
Margin="10" />
<Button Text="Bouton"
     VerticalOptions="Start"
     HorizontalOptions="Center" />
<Editor
     VerticalOptions="CenterAndExpand"
     HorizontalOptions="FillAndExpand"
Text="Lorem ipsum dolor sit amet, "
Margin="10" />
<Label Text="Pied de page"
VerticalOptions="End"
HorizontalOptions="End"
Margin="10" />
</StackLayout>
```


Capture 19 : Disposition linéaire avec Xamarin.Forms

RelativeLayout

Ce conteneur permet de disposer les contrôles les uns par rapport aux autres, en indiquant des contraintes. Ces contraintes peuvent porter :

- Sur la largeur ou la hauteur
- Sur la position en abscisse (X) ou en ordonnées (Y)

L'expression des contraintes est assez complète et peut permettre d'exprimer des choses assez complexes.

Prenons un exemple : nous commençons par placer un *label* que l'on souhaite centrer, en haut de l'écran. Il faut donc indiquer que son alignement en X est centré, et que l'on souhaite contraindre sa largeur à être égale à celle de son parent :

```
<Label x:Name="titre" Text="Titre !" Margin="10"
XAlign="Center"
RelativeLayout.WidthConstraint=
"{ConstraintExpression Type=RelativeToParent, Property=Width,
    Factor=1}" />
```

Imaginons ensuite une zone de couleur qui doit être placée en-dessous du titre (au moins 30 pixels) mais occuper tout l'espace disponible :

```
<BoxView x:Name="rect" Color="AntiqueWhite"

    RelativeLayout.YConstraint="{ConstraintExpression
    Type=RelativeToView, Property=Y, ElementName=titre,
    Constant=30}"

    RelativeLayout.WidthConstraint="{ConstraintExpression
    Type=RelativeToParent, Property=Width}"

    RelativeLayout.HeightConstraint="{ConstraintExpression
    Type=RelativeToParent,Property=Height}"
                 />
```

Et si nous souhaitons placer une zone de saisie à l'intérieur de cette zone :

```
<Editor Text="Lorem ipsum dolor sit amet..."
    TextColor="Black"

    RelativeLayout.XConstraint="{ConstraintExpression
    Type=RelativeToView, ElementName=rect, Property=X,
    Constant=20}"

    RelativeLayout.YConstraint="{ConstraintExpression
    Type=RelativeToView, ElementName=rect, Property=Y,
    Constant=20}"
                    />
```

Titre !

Lorem ipsum dolor sit amet...

Capture 20 : Disposition relative avec Xamarin.Forms

ScrollView

Ce type de disposition est assez pratique si le contenu est amené à dépasser l'écran. Couramment, à l'intérieur de ce conteneur sera placé un `StackLayout` de même orientation.

```
<ScrollView>
     <StackLayout Orientation="Vertical">
          <Label Text="Titre" HorizontalOptions="Center"
     VerticalOptions="Start" FontSize="Large"/>
        <Editor Text="Lorem ipsum dolor sit amet…"
          VerticalOptions="FillAndExpand"
          HorizontalOptions="FillAndExpand"  />
        <Button Text="Fermer" HorizontalOptions="Center"
     VerticalOptions="End" />
     </StackLayout>
</ScrollView>
```

Capture 21 : Disposition défilante avec Xamarin.Forms

Grid

La grille est le moyen le plus général de placer les contrôles.

Il faut commencer par définir une grille, c'est-à-dire préciser le nombre et la taille des colonnes et/ou des lignes.

Les tailles s'expriment en pixels ou peuvent être égales à « * » (taille automatique).

La grille suivante contient 3 colonnes dont une au centre d'une taille de 10 pixels, et 4 lignes dont la première et la dernière valent 50 pixels :

```
<Grid.ColumnDefinitions>
     <ColumnDefinition Width="*"/>
     <ColumnDefinition Width="10"/>
     <ColumnDefinition Width="*"/>
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
     <RowDefinition Height="50" />
```

```
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
    <RowDefinition Height="50" />
</Grid.RowDefinitions>
```

Plaçons ensuite dans cette grille des contrôles divers. Pour chaque contrôle, nous pouvons préciser :

- `Grid.Row` indique l'indice de la ligne où se place le contrôle
- `Grid.Column` indique l'indice de la colonne où se place le contrôle
- `Grid.RowSpan` indique sur combien de lignes (1 par défaut) s'étend le contrôle
- `Grid.ColumnSpan` indique sur combien de colonnes (1 par défaut) s'étend le contrôle

```
<Label Grid.Row="0" Grid.Column="0" Text="00"
    HorizontalOptions="Center"/>
<Label Grid.Row="0" Grid.Column="2" Text="02"
    HorizontalOptions="Center" />
<Label Grid.Row="1" Grid.Column="0" Text="Etiquette :"
    HorizontalOptions="End"/>
<Entry Grid.Row="1" Grid.Column="2"
    HorizontalOptions="CenterAndExpand" Placeholder="saisir
    votre texte" />
<BoxView Grid.Row="0" Grid.RowSpan="4" Grid.Column="1"
    Color="Gray"/>
<Button Text="Fermer" Grid.Row="3" Grid.Column="0"
    Grid.ColumnSpan="3" HorizontalOptions="Center" />
```

i) Les dispositions avec Qt

Qt possède deux méthodes pour définir des IHM. La première, les *widgets*, est la plus simple. Elle utilise un éditeur graphique intégré à *QtCreator*. La deuxième, plus récente et plus complexe, nécessite un langage spécial d'IHM appelé QML (mais peut également utiliser un éditeur graphique).

Nous utiliserons ici la version « widgets ».

Tous les contrôles Qt sont des classes qui héritent de `QWidget`. Qt fournit des dispositions à travers des classes héritant de `Layout` :

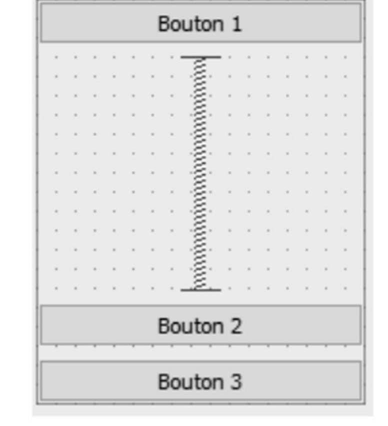
- `VerticalLayout` est une disposition en ligne d'orientation verticale
- `HorizontalLayout` est une disposition en ligne d'orientation horizontale
- `GridLayout` est une disposition en grille
- `FormLayout` est une disposition de type « formulaire » qui s'apparente à une disposition relative.

Les *layouts* peuvent être imbriqués les uns dans les autres.

La fenêtre peut ne pas avoir de *layout*, ce qui correspond à une disposition absolue.

VerticalLayout

Les différents contrôles sont disposés les uns en dessous des autres. Des marges peuvent être appliquées ou des contrôles de type « séparateur ». Chaque contrôle possède des propriétés qui indiquent sa « politique » de taille (`sizePolicy`) :



Capture 22 : disposition verticale avec Qt

- `Minimum` : le contrôle occupe la place minimale (suivant son contenu)
- `Fixed` : le contrôle occupe une taille fixée (en pixels en général)
- `Preferred` : le contrôle occupe si possible la taille préférée
- `Expanding` : le contrôle occupe le plus d'espace possible

Dans le *layout* précédent, nous avons placé trois boutons et un séparateur vertical. Chaque bouton possède la politique[1] suivante :

- Horizontale : minimum
- Verticale : fixée

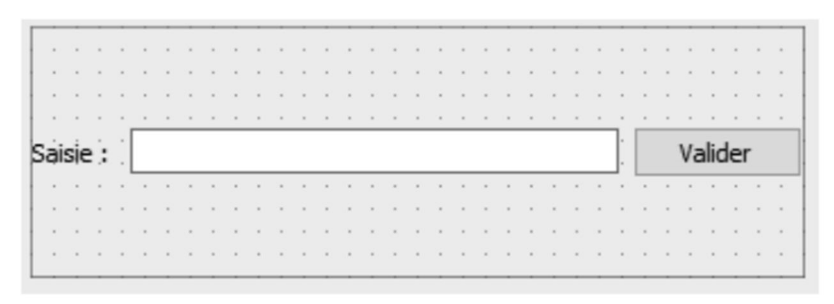
---

[1] Un composant Qt Widgets possède des caractéristiques définissant la manière dont il se comporte dans son conteneur : taille fixe, minimale, maximale, la plus grande possible…

HorizontalLayout

Similaire au précédent, mais avec une orientation horizontale et non verticale.

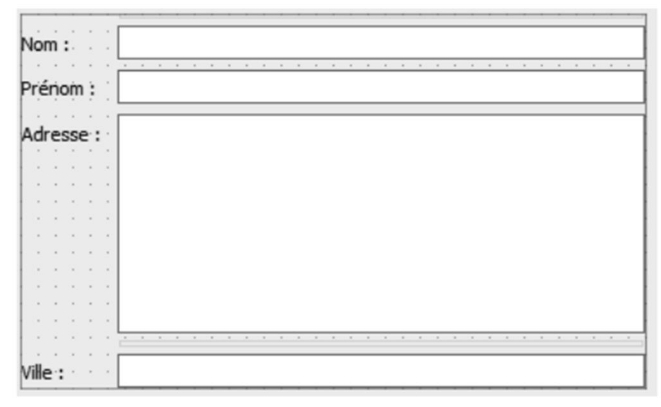
Par exemple, dans le *layout* horizontal précédent, un bouton (largeur minimale), une zone de saisie (largeur étendue) et un label (largeur minimale) :



Capture 23 : disposition horizontale avec Qt

FormLayout

Cette disposition est idéale pour un **formulaire**, comportant par exemple des ensembles étiquettes-zones de saisie et des boutons de contrôle.



Capture 24 : disposition formulaire avec Qt

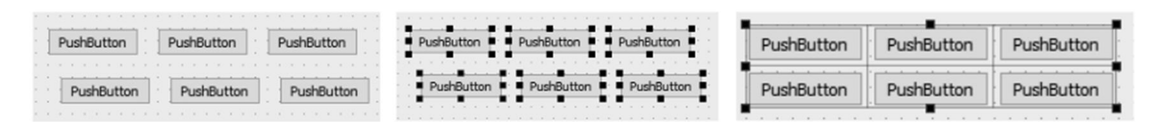
Voici l'exemple d'un *layout* de formulaire comportant des étiquettes (politique horizontale minimum et politique verticale fixe), des zones de saisies monoligne (politique horizontale étendue et politique verticale fixe) et une zone de saisie multiligne (politique horizontale étendue et politique verticale étendue).

GridLayout

La disposition en **grille** est appropriée dans la majorité des cas. Cette disposition n'est pas aussi pratique à utiliser que les autres et avec l'outil visuel il est difficile de placer exactement aux lignes/colonnes souhaitées les différents contrôles.

Un moyen simple de créer la grille est de placer les contrôles de manière absolue, puis de les sélectionner et de transformer l'ensemble en grille.

Dans l'exemple suivant, quelques boutons sont placés à peu près comme dans un tableau sur la fenêtre, puis sélectionnés, et clic droit → mettre en page → mettre en page dans une grille :



Capture 25 : disposition en grille avec Qt

Il est ensuite simple d'ajouter des lignes, colonnes, etc. dans le *layout*.

j) Les dispositions avec Cordova

Cordova utilise HTML et CSS pour définir l'IHM ; on retrouvera donc pour les dispositions, les caractéristiques propres à ces langages. Le but de cet ouvrage n'étant pas le développement web, un livre consacré à HTML/CSS sera plus pertinent pour établir toutes les caractéristiques. Voir (8) pour plus de détails.

Toutes les balises HTML affichables utilisent la balise CSS `display` qui permet de préciser la disposition utilisée. Les valeurs principales sont, en CSS3 :

- `inline` : l'élément est disposé en ligne à la suite du précédent
- `block` : l'élément est disposé comme un bloc (un rectangle) suivant le précédent
- `list-item` : disposition en liste
- `table` : disposition comme un tableau

Il est également possible de préciser la propriété `float` d'un élément pour régler, par exemple, l'habillage d'une image par rapport au texte :

- `left` : l'élément flotte à gauche de son parent
- `right` : l'élément flotte à droite de son parent
- `none` : l'élément ne flotte pas, il se positionne simplement

Disposition en ligne horizontale

```
<div>
    <label >Ceci est un texte</label>
    <button >Ajouter</button>
    <button >Modifier</button>
    <button >Supprimer</button>
    <input type="text"/>
</div>
```

Ceci est un texte | Ajouter | Modifier | Supprimer |

Disposition en ligne verticale

Chaque élément se place en-dessous du précédent. Il suffit de régler la propriété `display` à `block`, comme dans l'exemple suivant.

```
<div class="bloc" id="bloc1">
    <label >Ceci est un texte</label>
    <button >Ajouter</button>
    <button >Modifier</button>
    <button >Supprimer</button>
    <input type="text"/>
</div>
```

Ceci est un texte
Ajouter
Modifier
Supprimer

Fichier CSS lié:

```
.bloc *{
    display:block;
    margin-left:auto;
    margin-right:auto;
    margin-top:4px;
    margin-bottom:4px;
    width:90%;
}
#bloc1{
    width:160px;
    border : 1px solid black;
}
```

Disposition absolue

Le mode n'est pas pratique en HTML/CSS et peu approprié à des dimensions très variables. Il n'est donc pas conseillé. Si néanmoins vous voulez le faire, il faudra tout placer en « flottant » et régler les position, largeur et hauteur de chaque élément. Bonne chance !

4. Programmation évènementielle

Dès qu'on programme une IHM, une interaction avec l'utilisateur, on utilise de la programmation évènementielle.

En effet, l'IHM ne va pas effectuer ses traitements en séquence, comme un programme « classique », mais en réaction à des **évènements** qui sont issus de l'utilisateur et/ou du système.

Quelques exemples d'évènements utilisateur :

- Clic sur un bouton
- Pression du doigt sur un contrôle
- Déplacement du doigt (glissade)
- « Pincement » à deux doigts (effet de zoom)

Quelques exemples d'évènements système :

- L'application passe en arrière-plan
- Un certain temps s'est écoulé depuis le démarrage

Faire de la programmation évènementielle revient donc à écrire du code en réponse à des évènements. La méthode/technique employée dépend fortement du langage/*framework* utilisé.

a) Le patron observateur

La plupart des *frameworks* pour l'IHM utilisent le patron de conception (*design pattern* en anglais) observateur (1) pour la gestion des évènements. Ce patron est utilisé quand un objet doit être prévenu du changement d'état d'un autre objet lors d'un évènement déclencheur, tout en ne connaissant pas à l'avance l'objet. Il est donc tout indiqué en programmation évènementielle.

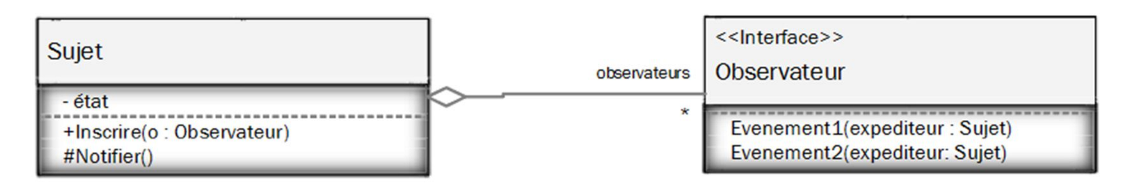
Son principe de base consiste à utiliser deux objets : un « sujet » ou « observable » qui peut changer d'état, et un « observateur » qui doit être prévenu du changement d'état.

L'observateur est une **interface** qui contient, en opération, les différents évènements pouvant arriver (avec des paramètres éventuels).

Le sujet (ou observable) est une interface ou une classe qui connait un (ou plusieurs) observateur(s) et qui va donc, en cas de changement d'état ou de déclenchement d'un évènement, les prévenir via une opération de leur interface.

Une représentation UML classique du patron est la suivante :

Diagramme 10 : diagramme de classes pattern observateur



Une classe voulant être prévenue des changements d'état du sujet doit donc implémenter `Observateur` et s'inscrire auprès du sujet. Quand le sujet change d'état, via son opération `Notifier`, il appellera les opérations *ad hoc* de(s) observateur(s) inscrit(s).

b) Java / Android

Java utilise également le patron observateur. Le mot utilisé est `Listener` (écouteur, en anglais) pour parler d'observateur. Sous Android, le principe est le même.

Les évènements sont regroupés par famille (exemple : actions de l'utilisateur sur un bouton) dans une interface. Chaque évènement est en fait une opération de cette interface. Pour répondre à un évènement, une activité doit donc :

- Implémenter l'interface correspondante,
- S'inscrire auprès du système comme « écouteur »,
- Répondre à l'évènement souhaité dans les opérations héritées.

Prenons par exemple le cas de l'évènement « clic sur un bouton ». L'interface correspondant à cet évènement est `OnClickListener` :

notre activité doit donc implémenter cette interface, ce qui lui impose de définir une opération `onClick` prenant une instance de `View` en paramètre. Le paramètre permet de savoir quel contrôle a été cliqué, ce qui est pratique car si l'activité possède plusieurs boutons il faut pouvoir les différencier ! Il faut également que l'activité soit inscrite auprès du bouton comme écouteur…

Imaginons une activité possédant dans son *layout* un bouton dont l'identifiant est `button` :

```
<Button android:id="@+id/button" … />
```



La classe de l'activité doit implémenter l'interface :

```
class MyActivity extends Activity implements OnClickListener
```

Et donc avoir une opération `onClick` (cette opération contient le code devant être exécuté en réponse au clic du bouton) :

```
@Override
public void onClick(View view) {…}
```



Dans le `onCreate`, il faut relier le bouton et l'activité en « inscrivant » celle-ci à l'évènement :

```
Button button = (Button)findViewById("button") ;
button.setOnClickListener(this) ;
```



c) Swift / iOS

Swift est d'assez haut niveau et « cache » pas mal de choses au développeur. Ce qui est souvent, il faut bien l'avouer, plus simple pour nous. Dans le cas des évènements, tout se passe à la souris ou presque. Comme vu lors du développement du « hello world » en Swift (voir page 55), il suffit de ctrl-glisser un contrôle (bouton par exemple) sur le code du contrôleur, de créer une connexion de type « action » en choisissant bien l'évènement (*event*) voulu (par exemple : *Touch Up Inside*). Une opération correspondante est automatiquement créée dans la classe Swift.

d) C# / Windows / Xamarin

C# utilise une notion très proche du patron observateur vu plus haut. Un évènement est un type particulier et il est associable à une opération

particulière appelée un délégué (*delegate*). Le prototype de l'opération est rendu obligatoire par le type de délégué. En *Xamarin.Forms*, par exemple, tous les évènements utilisateur possèdent un paramètre de type `Object` (l'objet ayant émis l'évènement) et un paramètre de type `EventArgs` (les éventuels paramètres de l'évènement, qui dépendent du type d'évènement).

Pour relier un évènement avec sa réponse, il suffit d'utiliser l'opérateur `+=` qui sert à inscrire une opération (un délégué) auprès d'un évènement. L'opérateur inverse `−=` existe également, il sert à désinscrire l'opération. Ce lien peut se faire dans le code ou directement dans le fichier XAML. Dans ce cas, le lien est fait par une chaîne de caractères correspondant à l'opération de la classe.

Prenons par exemple un bouton, en *Xamarin.Forms*, dont l'évènement `OnClick` doit être relié à une opération `Clic` de notre classe de page. L'opération doit être comme suit :

```
private void Clic(object sender, EventArgs args)
{
// …
}
```

Le lien peut être intégré directement dans le fichier `.xaml` de la page, comme suit :

```
<Button x:Name="button1" OnClick="Clic" />
```

Mais il est également possible de réaliser le lien dans le code C# de la page :

```
button1.OnClick += Clic ;
```

e) C++ / Qt

C++ ne possède pas d'usage particulier pour les évènements, puisque le langage ne contient aucun *framework* pour l'IHM. Qt, en revanche, prévu au départ pour les IHM, fournit un mécanisme basé sur les **signaux** et les ***slots***.

Un signal Qt est un évènement émis par un objet Qt (`QObject`), en général un composant du *framework* (un bouton, une fenêtre, etc…) qui joue le rôle de sujet.

Un *slot* Qt est une opération d'un objet Qt (`QObject`), en général une classe de fenêtre qui joue le rôle d'observateur.

La liaison entre les deux peut être effectuée automatiquement par le concepteur graphique, ou par code.

Prenons par exemple le cas d'une application Qt où un bouton est placé sur une fenêtre. Pour créer dans cette fenêtre le *slot* relié au bon signal, il suffit de faire un clic-droit sur le bouton dans l'éditeur graphique, et choisir le signal : le *slot* sera automatiquement créé.

Prenons un autre exemple : un évènement régulier lié à un *timer* (classe `QTimer`). Le concepteur graphique ne peut pas réaliser la liaison directement. Dans la classe de fenêtre, il faut déclarer un *slot* pour le *timer* :

```
private slots :
    void maj() ;
```

Pour relier le signal du *timer* avec ce *slot*, il faut utiliser la macro `CONNECT` :

```
timer = new QTimer ;
connect(timer, SIGNAL(timeout()), SLOT(maj())) ;
```

L'éditeur de Qt Creator fournit automatiquement les signaux et les *slots* disponibles en auto-complétion.

f) Javascript / Cordova

En Javascript, un évènement est un type d'objet particulier qui est une fonction… Il est donc possible d'associer à un évènement une fonction soit déjà existante, soit créée en ligne directement.

Prenons par exemple l'évènement `onload` de l'objet `window`, qui est exécuté lorsque la page est totalement chargée. Pour répondre à l'évènement, il suffit de créer une fonction en ligne comme dans l'exemple ci-dessous :

```
window.onload = function(){
    // faire quelque chose
} ;
```

Il est possible bien entendu d'affecter une fonction existante, comme dans l'exemple ci-dessous :

```
window.onload = OnLoad ;
//…
function OnLoad(){
     // faire quelque chose
}
```

La première forme est plus compacte mais parfois plus difficile à lire.

Un évènement non affecté possède la valeur `undefined`.

5. Changement d'écran

Il est rare qu'une application ne comporte qu'un seul écran… même si cela arrive, dans les cas des applications simples, ou dans le cas des écrans « à onglets » où le même écran peut présenter plusieurs groupes de contrôles. Néanmoins il arrive que l'on ait besoin de passer d'une page à l'autre, éventuellement en transmettant des informations. Bien entendu, le changement d'écran est totalement dépendant de l'outil utilisé (et la plateforme).

Nous allons étudier dans la suite de cette partie le passage d'une écran `First` à un écran `Second`, pour chacun des outils utilisés. Nous allons faire une petite variable du « hello, world » pour cela. Le premier écran comprendra une zone de saisie pour écrire un prénom et un bouton pour passer à l'écran suivant. L'écran suivant contiendra le texte « bonjour » suivi du prénom saisi, qui devra donc être passé en paramètre.

a) Android

Un écran Android est une activité, donc une classe Java. Pour passer d'une activité à l'autre il faut utiliser la méthode `startActivity` de la classe `Activity`.

Il faut commencer par créer une instance de la classe `Intent` liée au type de la 2e activité. Dans l'opération de réponse au clic du bouton de la 1ère activité, nous trouverons donc :

```
Intent intent = new Intent(this, Second.class) ;
```

Une fois `l'intent` créé, il est possible de lui ajouter des paramètres, appelés « *extra* » par Android. Ces paramètres peuvent être des entiers, des

flottants, des booléens, des chaînes, ou tout `Object` sérialisable. Les paramètres sont identifiés par un nom donné sous la forme d'une chaîne.

Ici, si nous voulons transmettre le texte contenu dans un `EditText` nommé `nom` :

```
intent.putExtra("nom",nom.getText()) ;
```

Pour démarrer l'activité :

```
startActivity(intent) ;
```

La deuxième activité passera donc au premier plan.

Afin de récupérer la valeur passée, pour par exemple l'afficher dans un `TextView` appelé `message`, il faut ajouter, à la fin de l'opération `onCreate` :

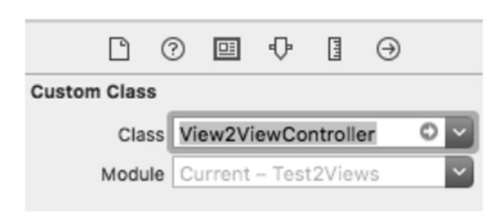
```
String param = getIntent().getExtras().getString("nom") ;
message.setText(param) ;
```

b) iOS

XCode propose une vue appelée *storyboard* pour indiquer la succession des écrans. Il n'est pas obligatoire de l'utiliser mais avec un nombre limité d'écrans, c'est assez simple.

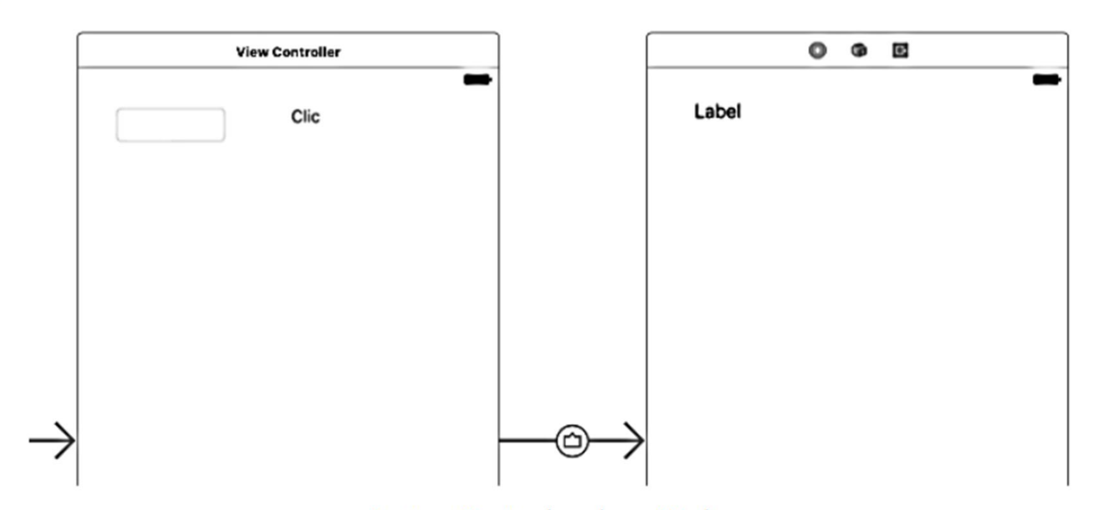
Il faut commencer par faire la première vue (il n'y en a qu'une au départ…). Par exemple nous pouvons y déposer une zone de saisie et un bouton. N'oubliez pas de relier ces contrôles au code du contrôleur par un ctrl+glisser.

Il faut ensuite créer un 2e contrôleur de vue : ajoutez un fichier de type `Cocoa Touch Class`, lui choisir le type `UIViewController`. Créez ensuite une deuxième vue en effectuant un glisser-déposer d'un contrôle de type `ViewController` depuis les outils (à droite en général). Cette vue peut contenir par exemple un *label*. Indiquez dans les propriétés du contrôleur qu'il est du type créé précédemment.





Pour indiquer que le clic sur le bouton déclenche le changement de vue, il suffit de CTRL+glisser le bouton sur la 2e vue. Un menu s'affiche ensuite dans lequel on choisit le mode d'ouverture (par exemple : `present modally`). Une flèche apparait alors sur le *storyboard* pour indiquer le passage d'une vue à l'autre. C'est simple et sans code, mais comment passer des paramètres à la deuxième vue ?



Capture 26 : storyboard sous XCode

Une possibilité est d'utiliser une propriété de la 2e vue et de redéfinir, dans la 1ère vue, la méthode `prepare` qui sera appelée avant de quitter celle-ci.

Dans la classe du 2e contrôleur, il faut déjà ajouter une propriété :

```
var param=""
```

Toujours dans le 2e contrôleur, voici comment utiliser cette propriété pour modifier le *label* de la vue :

```
override func viewDidLoad()
{
    super.viewDidLoad()
    message.text = param
}
```

Swift

Et enfin, dans le premier contrôleur, il ne reste qu'à définir la méthode `prepare` pour « préparer » le passage d'une vue à l'autre :

```
override func prepare(for seque :UIStoryboardSeque,
     sender : Any ?)
{
     let sec = seque.destination as! View2ViewController
     sec.param = nom.text!
}
```

c) Windows

Les écrans sont de type `Page`. Pour changer de page, on utilise la méthode `Navigate` de l'objet `Frame`, en précisant le type de la page et l'éventuel paramètre. Le paramètre étant de type `Object`, il peut être de n'importe quel type (pas besoin d'être sérialisable).

Exemple pour accéder à une page de type `SecondPage` en passant en paramètre la chaîne contenue dans un `TextBox` `nom` :

```
Frame.Navigate(typeof(SecondPage), nom.Text);
```

On remarque que la deuxième page n'est pas construite de manière classique (en appelant son constructeur) mais par le système de navigation. Pour récupérer le paramètre, il faut donc dans la classe `SecondPage` redéfinir la méthode `OnNavigatedTo` qui sera appelée lors de la navigation. Le paramètre de cette méthode contient, entre autres, le paramètre passé. Exemple de code qui récupère le paramètre passé et l'affiche dans un `TextBlock` appelé `message` :

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
     String nom = e.Parameter as String;
     message.Text = "Bonjour " + nom;
}
```

Pour revenir à la page précédente, il est possible pour l'utilisateur d'appuyer sur le bouton retour (s'il est présent) mais il est également possible de le faire par code en appelant la méthode `GoBack` de l'objet `Frame` :

```
Frame.GoBack() ;
```

d) Xamarin.Forms

L'écran en *Xamarin.Forms* est une classe héritant de `Page`. Le système de navigation entre pages est très proche de Windows.

Pour passer d'une page à l'autre, on crée une instance de la deuxième page (en appelant son constructeur, qui peut donc avoir des paramètres) et la passer en paramètre à l'opération `PushModalAsync` de l'objet `Navigation`.

Exemple avec l'ouverture d'une page de type `SecondPage` en lui transmettant en paramètre le contenu d'une zone de saisie appelée `nom` :

```
Page p = new SecondPage(nom.Text);
await Navigation.PushModalAsync(p);
```

La récupération du/des paramètre(s) se fait très simplement dans le constructeur de la page. Exemple avec la récupération du nom passé en paramètre et affiché dans un *label* appelé `message` (attention : il faut bien le faire après l'appel de la fonction `InitializeComponent`, sinon le *label* n'est pas initialisé) :

```
public SecondPage (string nom)
{
     InitializeComponent ();
     message.Text = nom;
}
```

Pour revenir à la page précédente, l'utilisateur peut bien entendu utiliser le bouton retour arrière (s'il est présent) mais il est possible de le faire par code en appelant la méthode `PopModalAsync` de l'objet `Navigation` :

```
await Navigation.PopModalAsync() ;
```

e) Qt

Avec *QWidgets*, chaque écran est une instance d'une classe héritée de `QWindow`. La fenêtre principale est de type `QMainWindow` et une fenêtre secondaire est en général de type `QDialog`. Le passage à une deuxième fenêtre se fait tout simplement en créant la fenêtre (donc appel de son constructeur, lequel peut avoir des paramètres) puis en appelant sa

méthode show (cas non modal[1]) ou exec (cas modal). Exemple avec une première page contenant une zone de saisie ainsi qu'un bouton et transmettant à une deuxième page (type Second) le nom saisi :

```
Second page(this, ui->nom->text());
page.exec();
```

La récupération du(des) paramètre(s) s'effectue tout naturellement dans le constructeur, et le retour à la page précédente se fait en appelant la méthode close.

Exemple avec une fenêtre (type Second) qui récupère une chaine en paramètre et l'affiche dans un *label* nommé message :

```
Second::Second(QWidget *parent, QString n) :
    QDialog(parent),
    ui(new Ui::Second)
{
    ui->setupUi(this);
    ui->message->setText(n);
}
```

f) Cordova

Avec *Cordova* (donc en HTML) un écran est une page HTML. Pour passer d'une page à l'autre, il est possible soit de faire un lien (balise a dans le fichier HTML) mais dans ce cas il ne peut y avoir de paramètres déterminés par le code, soit d'utiliser la propriété href de l'objet location contenu dans l'objet window en *Javascript*.

Le problème du passage des paramètres se pose… Il n'y a que deux possibilités : soit passer par l'URL (en ajoutant après le nom du fichier le caractère « ? » suivi des paramètres) soit en utilisant un *cookie*[2].

Exemple avec l'utilisation d'un *cookie* pour transmettre à une 2e page (page2.html) la chaîne saisie par l'utilisateur dans une zone de saisie dont l'id est nom :

---

[1] Modal : il existe deux manières d'ouvrir une page/fenêtre fille : soit celle-ci « bloque » la fenêtre/page appelante (cas modal) soit les deux fenêtres sont indépendantes (cas non modal).

[2] Petit fichier laissé sur l'ordinateur exécutant le navigateur (et donc sur le mobile dans le cas de Cordova).

```
document.cookie = document.getElementById("nom").value;
window.location.href = "page2.html";
```



Pour récupérer le paramètre, on utilise, dans un fichier `.js` contenu dans `page2.html`, l'évènement `onload` de la fenêtre pour récupérer le *cookie* :

```
window.onload = function ()
{
    var nom = document.cookie;
    document.getElementById("message").innerHTML =
       "Bonjour " + nom;
}
```

Exemple avec l'utilisation des paramètres par l'URL (même principe que plus haut) :

```
var nom = document.getElementById("nom").value;
window.location.href = "page2.html?"+nom;
```

Pour récupérer le paramètre, il faut utiliser la propriété `search` de l'objet location situé dans l'objet `window`. Dans notre cas, qui est simple, il suffit de ne pas retenir le premier caractère (le `?`) mais dans le cas de plusieurs paramètres il faudrait un découpage plus complexe (en utilisant par exemple des expressions régulières). Exemple pour récupérer le nom dans l'URL :

```
var params = window.location.search;
var nom = params.substr(1, params.length - 1);
document.getElementById("message").innerHTML = "Bonjour " +
     nom;
```

VI. Les ressources

Une application pour téléphone comprend, en sus des différents écrans et du code, un certain nombre de données appelées **ressources**. Ces données peuvent être de natures diverses, les plus courantes étant le **texte** et les **images**.

Ces ressources sont incluses dans le *package* de l'application, distribuées avec elle et accessibles facilement depuis celle-ci. On peut les voir comme des fichiers d'accompagnement, invisibles à l'utilisateur et empaquetés dans l'application elle-même.

Ces ressources ont un intérêt supplémentaire : il est possible de spécifier dans quel contexte une ressource est utilisable. Suivant les systèmes, ces contextes sont :

- La **langue** : notamment les ressources « texte » peuvent être différentes suivant la langue du système
- La **résolution** et/ou la **taille** de l'écran : les images, par exemple, peuvent être différentes, adaptées à l'écran
- **L'orientation** (portrait ou paysage) de l'écran : ici encore, il est intéressant d'adapter les images !
- La **présence** ou non d'un capteur, d'un périphérique
- La **version** du **système** d'exploitation

1. Les ressources avec Android Studio

Dans un projet Android, un dossier est spécifiquement réservé aux ressources. Son chemin relatif au chemin du projet est `app/src/main/res`. Dans ce dossier, nous pouvons trouver les ressources suivantes (9) :

- `Layout` : fichiers XML représentant les vues du système
- `Drawable` : fichiers image (PNG, JPG ou GIF)
- `Values` : valeurs diverses (chaînes, couleurs, styles…)
- `Animator` et `Anim` : fichiers XML contenant les animations
- `Mipmap` : fichiers images pour les icônes du lanceur
- `Menu` : fichiers XML contenant les menus
- `Raw` : données quelconques

- `Xml` : données quelconques, au format XML
- `Font` : polices de caractères

a) Ressources différentes suivant les appareils

A ces dossiers peuvent être ajoutés un grand nombre de dossiers représentant les alternatives. Les alternatives permettent d'avoir des ressources différentes suivant la configuration de l'appareil qui exécute l'application. Android utilise une syntaxe simple pour un tel dossier : il ajoute au nom « classique » un ou plusieurs qualificateurs.

Les qualificateurs classiques sont les suivants (il en existe d'autres, qui sont moins utilisés) :

- Spécification de la **langue** et de la **région**. Suivant les codes ISO-639-1[1] et ISO-3166[2]. Exemple : `values-fr` pour les appareils en français, `values-es` pour les appareils en espagnol, `values-fr-rCA` pour les appareils en français canadien.
- **Direction** : on peut lire de gauche à droite (comme en français) ou de droite à gauche (comme en arabe). Deux possibilités : `ldrtl` (droite à gauche) et `ldltr` (gauche à droite). Exemple : `values-ldrtl` pour une écriture de droite à gauche.
- **Taille** d'écran : quatre tailles sont définies. Exemple : `drawable-large` pour les images sur les écrans de grande taille.
    - `small` : pour les petits écrans. Une montre par exemple.
    - `normal` : pour les écrans de taille normale, un téléphone par exemple
    - `large` : pour les écrans de grande taille, comme une tablette
    - `xlarge` : pour les écrans géants, comme un téléviseur.
- Orientation : deux orientations possibles : `port` (portrait) et `land` (paysage). Exemple : `drawable-land` pour les images en mode paysage.

---

[1] Code normalisé représentant les langues sur deux caractères.

[2] Code normalisé représentant une région du monde (pays par exemple) sur 2 caractères.

- Densité de l'écran : exprimée en DPI[1]. Les densités possibles sont, dans l'ordre de densité croissante : ldpi, mdpi, hdpi, xhdpi, xxhdpi, xxxhdpi. Un téléphone moyen de gamme actuel possède une densité de type hdpi.

Ces qualificateurs peuvent être utilisés ensemble. Le dossier drawable-land-hdpi, par exemple, concerne les images en densité haute avec une orientation paysage. Si la configuration de l'appareil n'est pas indiquée, alors c'est le dossier par défaut qui est utilisé.

Nous pouvons par exemple définir un layout différent (c'est une ressource) pour les modes portrait et paysage, ou pour un téléphone (taille normale) et une tablette (taille large), tout en ayant le même code de l'activité.

b) Utilisation des ressources texte : plusieurs langues

Prenons un exemple : nous avons dans une application Android des ressources de type texte, qui sont stockées dans res/values/strings.xml. Si nous voulons l'application en plusieurs langues, par exemple français, anglais et espagnol, il faut créer plusieurs dossiers de ressources. Mais il faut aussi penser aux appareils qui ne sont pas dans ces 3 langues : quelle est la langue par défaut ?

Les valeurs par défaut sont dans les dossiers « classiques » : res/values/strings.xml pour les textes.

Si nous souhaitons des textes en anglais par défaut, il faut mettre dans res/values/strings.xml les valeurs en anglais, puis dans res/values-fr/strings.xml les valeurs en français et dans res/values-es/strings.xml les valeurs pour l'espagnol.

Les ressources texte, définies dans res/values/strings.xml, sont exprimées sous la forme suivante, où id est une chaîne identifiant la ressource, et value la valeur de celle-ci :

```
<string name="id">value</string>
```



[1] *Dots Per Inch* : points par pouce (un pouce fait environ 2,54cm).

Pour accéder à la valeur de la ressource dans un `layout`, il suffit d'indiquer sa référence (ici `id`) :

```
<TextView android:text="@string/id" />
```

Pour accéder à la valeur dans le code Java (dans une activité), il suffit d'utiliser le code suivant (`id` étant l'identifiant de la ressource) :

```
String s = getResources().getString(R.string.id) ;
```

Nous allons voir un exemple pour internationaliser une application simple. Créez un projet avec Android Studio. Notre application sera un simple « *hello world* » mais avec deux langues possibles : français et anglais (l'anglais étant par défaut). L'application contiendra un message « Saisir votre nom », une zone de saisie, un bouton « valider » et donnera un message « bonjour *x* » avec *x* le nom saisi.

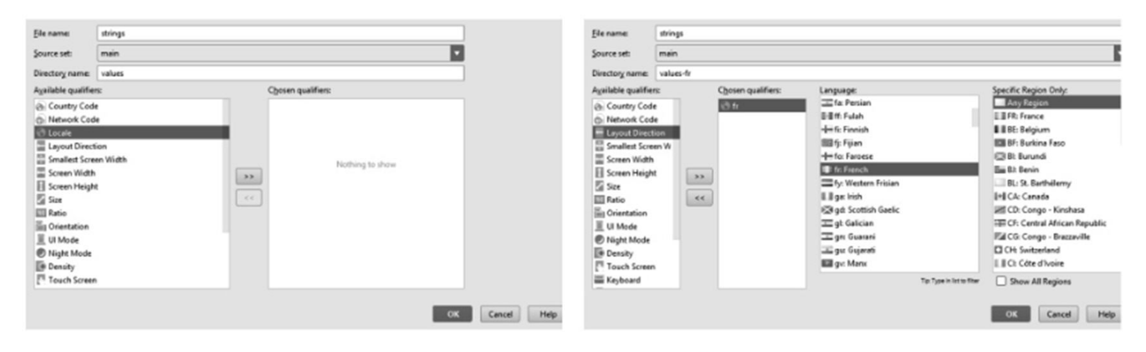
Nous allons commencer par créer nos ressources. Soient les textes suivants :

- Le message de bienvenue : « saisir votre nom » en français, et « *enter your name* » en anglais. Appelons cette ressource `message`.
- Le texte du bouton : « valider » en français, « *submit* » en anglais. Appelons cette ressource `valider`.
- Le message de bienvenue : « Bonjour » en français, « *Hello* » en anglais. Appelons cette ressource `bienvenue`.

Le projet contient par défaut un fichier `res/values/strings.xml`. Editez ce fichier et ajoutez les ressources en anglais (une ressource est déjà présente, le nom de l'application) :

```xml
<resources>
        <string name="app_name">Hello</string>
        <string name="message">Enter your name</string>
        <string name="valider">Submit</string>
        <string name="bienvenue">Hello</string>
</resources>
```

Il faut ensuite créer les ressources en français. Pour cela, le plus simple est d'utiliser les assistants d'Android Studio : faites un clic-droit sur le dossier `values`, et choisir « *new → values resource file* ». Choisir `strings` comme nom de fichier et `locale` comme qualificateur, puis `fr` pour le français :



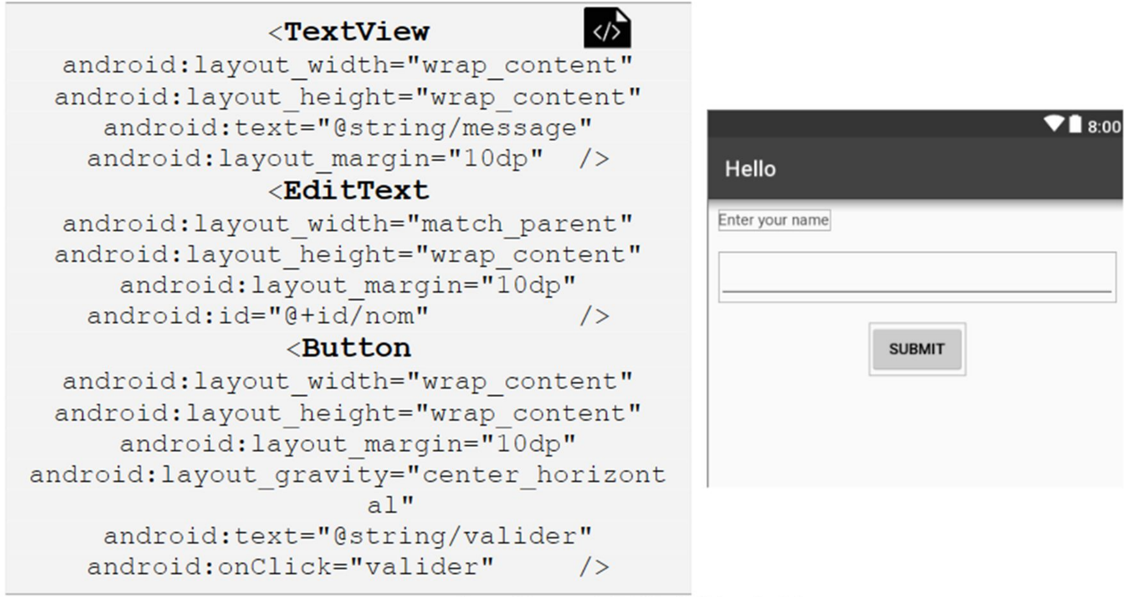
Capture 27 : création de ressources texte multi langues (Android)

Un fichier `strings.xml` spécifique au français est créé, vous pouvez entrer les valeurs suivantes :

```xml
<resources>
        <string name="app_name">Salut le monde</string>
        <string name="message">Saisir votre nom</string>
        <string name="valider">Valider</string>
        <string name="bienvenue">Bonjour</string>
    </resources>
```

Faites bien attention : les noms des ressources doivent être identiques !

Nous pouvons ensuite éditer le `layout` de l'activité principale (par défaut le fichier `activity_main.xml`) et placer une zone de texte (`TextView`), une zone de saisie (`EditText`) et un bouton (`Button`). Un `layout` horizontal (voir V.3.b) est ici indiqué :



```
<TextView
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="@string/message"
android:layout_margin="10dp"  />
<EditText
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:layout_margin="10dp"
android:id="@+id/nom"          />
<Button
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_margin="10dp"
android:layout_gravity="center_horizont
al"
android:text="@string/valider"
android:onClick="valider"     />
```


Capture 28 : disposition activité "hello" (Android)

Dans la classe de l'activité, il faut bien entendu ajouter un attribut pour la zone de saisie :

```
private EditText nom;
```

Il faut également l'initialiser dans le `onCreate` :

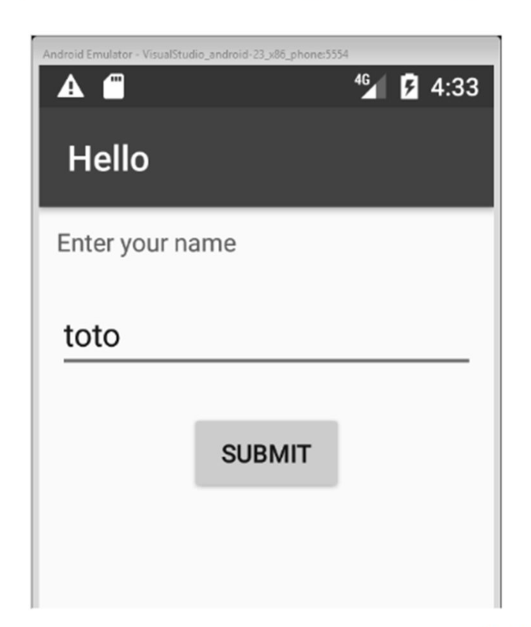
```
nom = (EditText)findViewById(R.id.nom);
```

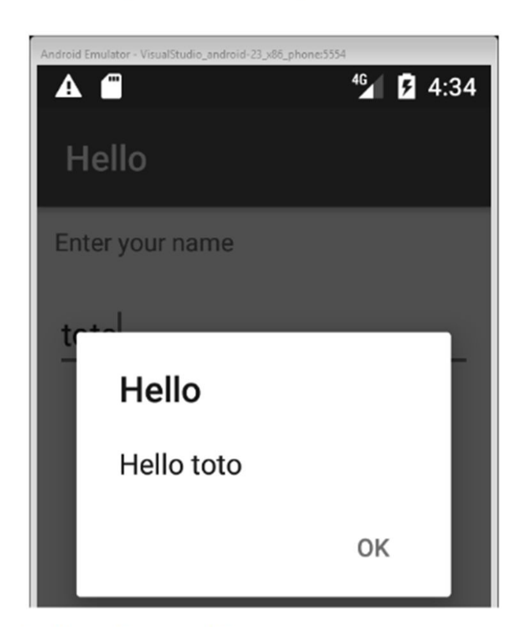
Il ne reste plus qu'à coder la fonction de réponse au clic sur le bouton. Celle-ci va ouvrir une petite boite de message avec un message différent en fonction de la langue :

```
public void valider(View v){
     String n = nom.getText().toString();
     String msg =
     getResources().getString(R.string.bienvenue)
           + " "+n;
     AlertDialog.Builder builder = new
     AlertDialog.Builder(this);
     builder.setTitle(R.string.app_name)
        .setMessage(msg)
        .setPositiveButton("Ok",
           new DialogInterface.OnClickListener() {
              @Override
              public void onClick(DialogInterface
     dialogInterface,
                 int i) { dialogInterface.cancel();        }
```

```
            });
        AlertDialog dialog = builder.create();
        dialog.show();
}
```

L'exécution de l'application dans l'émulateur donne les résultats suivants (en modifiant la langue de l'appareil dans les préférences) :





Capture 29 : "hello" Android, système anglais





Capture 30 : "hello" Android, système français

Cet exemple nous montre qu'il faut éviter d'indiquer le texte directement dans le code Java ou dans le XML, mais qu'il vaut mieux utiliser les

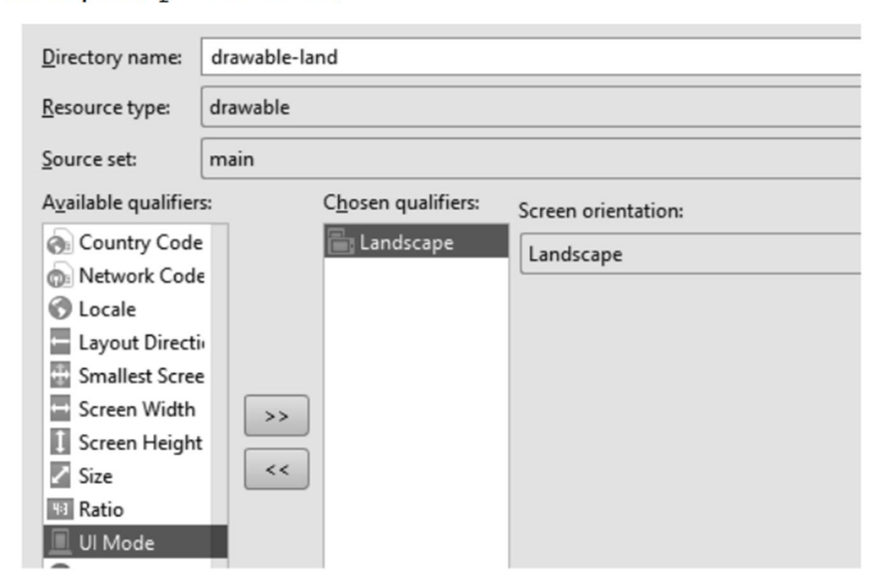
ressources, ce qui facilite l'écriture d'applications en plusieurs langues. Le surcoût de code et/ou d'XML est négligeable.

c) Utilisation des ressources images

Nous allons créer une application simple qui affichera une image de fond différente en orientation portrait et en orientation paysage.

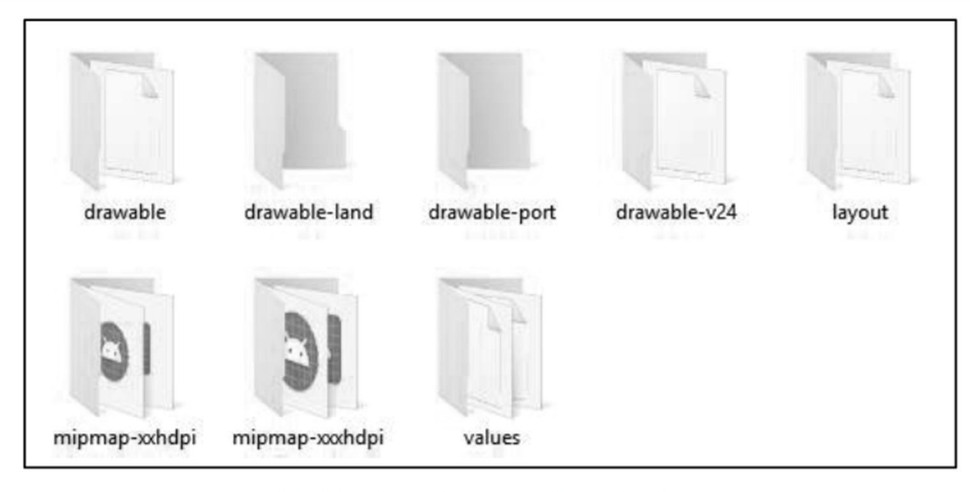
Créez une application avec Android Studio. Dans le `layout` de l'activité principale, placez un simple `ImageView` qui occupera tout l'écran.

Choisissez deux images (personnelles, récupérées sur Internet, peu importe) d'orientation différente. Donnez-leur le même nom (par exemple : `image.png`) Créez un répertoire pour l'orientation paysage et un autre pour l'orientation portrait ; pour ce, il suffit de faire un clic-droit sur le dossier `res` et de choisir « *new → Android resource directory* ». Choisissez le type `drawable`, le qualificateur `Orientation`, puis `landscape`. Recommencez pour `portrait`.



Capture 31 : création d'un dossier de ressources visuelles Android

Observez votre dossier : vous allez trouver dans `res` les dossiers suivants :



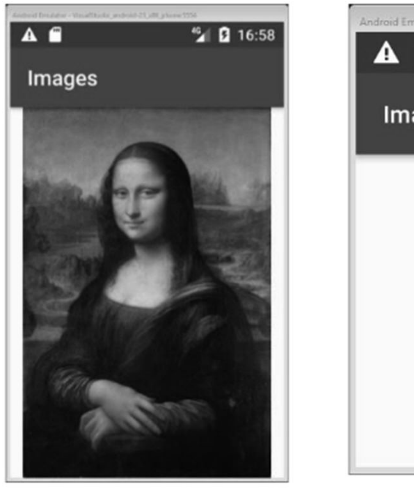
Capture 32 : dossiers ressources Android

Placez l'image en paysage dans `drawable-land` et l'image en portrait dans `drawable-port`. Si vos images s'appellent `fond.png`, la ressource image s'appelle `fond`.

Il ne reste plus qu'à indiquer dans le `layout` la source de l'image comme ressource :

```xml
<ImageView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:src="@drawable/fond" />
```

Lors de l'exécution, si on modifie l'orientation (on tourne son téléphone ou l'émulateur) l'image va changer :



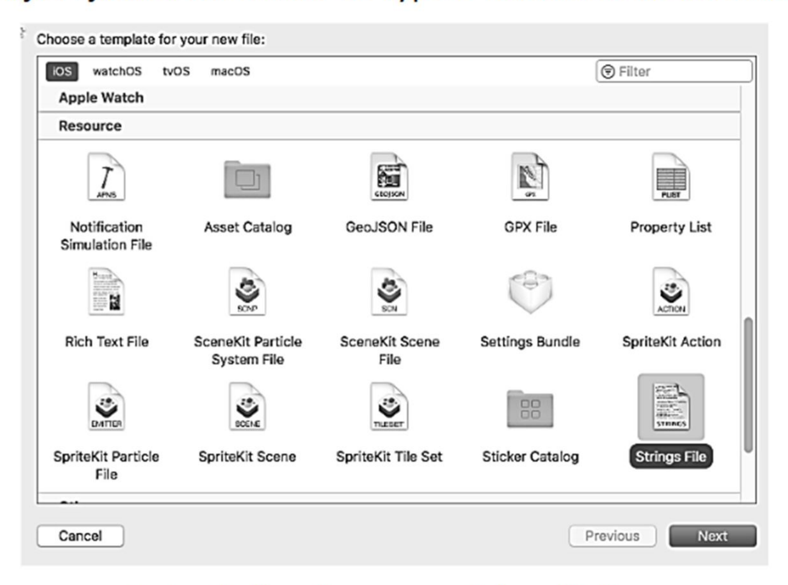


Capture 33 : changement automatique du fond selon l'orientation (Android)

2. Les ressources avec XCode

a) Ressources texte

Dans le projet ajoutez un fichier de type « ressource de chaînes » :



Capture 34 : Nouvelle ressource « chaînes » XCode

Il faut ensuite donner un nom au fichier de chaînes. Ce type de fichier étant fréquemment utilisé pour la localisation des applications, il existe des fonctions pour lire les chaînes depuis le fichier `Localizable.strings`. Il est donc conseillé de donner le nom `Localizable` à ce fichier.

Un fichier de chaînes contient simplement des paires *nom-valeur*. Chaque paire comprend une clé (chaîne), le symbole =, une valeur (chaîne) et se termine par un point-virgule :

```
"Title"="Languages app test" ;
"Content"="This app is only made to test Strings in XCode" ;
```

Dans le code Swift, pour utiliser une ressource texte, il suffit d'utiliser la fonction `NSLocalizedString` en précisant la clé cherchée (le deuxième paramètre est un commentaire, inutile ici) :

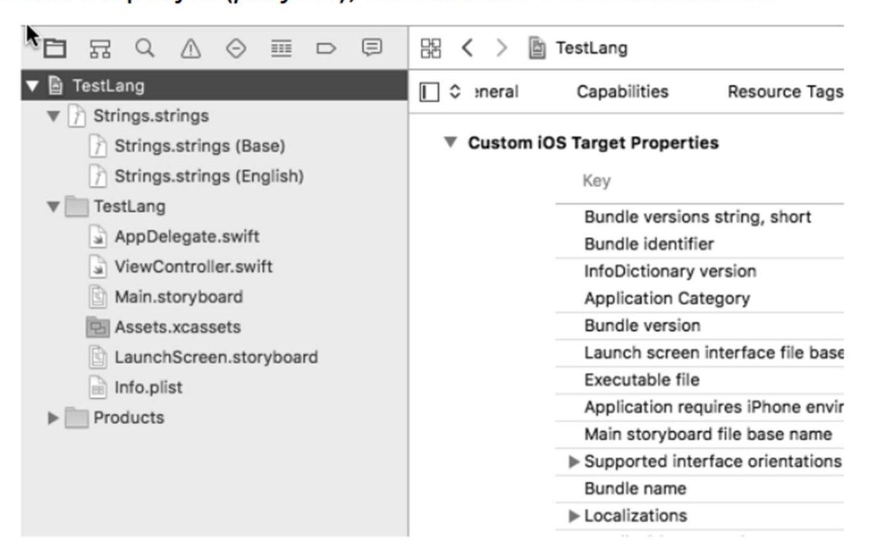
```
let texte = NSLocalizedString("Content",comment :"")
```

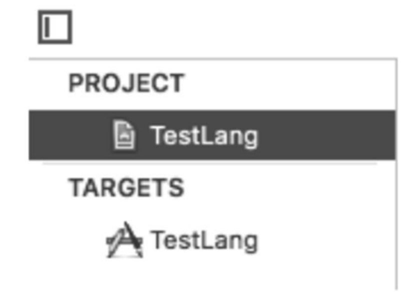
Le *storyboard*, cependant, ne peut utiliser directement ces ressources chaînes. Il est nécessaire de passer par le code, ou d'utiliser l'internationalisation du *storyboard* (plusieurs versions du même *storyboard*

dans des langues différentes) qui est un mécanisme similaire. Nous verrons un peu plus loin comment localiser le *storyboard*.

Les ressources « texte » sont fréquemment utilisées pour les messages dans l'application car celles-ci peuvent être « localisées », c'est-à-dire dépendantes de la localisation / langue de l'appareil.

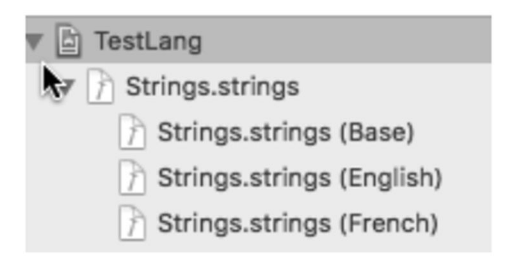
Il faut commencer par choisir, dans le projet XCode, quelles sont les langues utilisées par l'application. Pour cela, nous devons sélectionner le projet et la partie « *Localizations* ». Attention : quand on sélectionne le projet, il est fréquent que les paramètres apparaissant soient ceux de la cible (*target*) et non du projet (*project*), comme sur l'écran suivant :





Si c'est le cas, il suffit de cliquer sur le symbole et de choisir le projet. Sous la partie « *Localizations* », ajoutez les langues souhaitées. Une fenêtre apparait alors pour choisir quels fichiers du projet doivent être « localisés » (différents suivant la langue). Ne choisissez que les ressources chaînes pour l'instant.

A présent, notre fichier de ressource chaînes contient trois versions : la version dite « *base* », la version « *english* » et la version « *french* » :

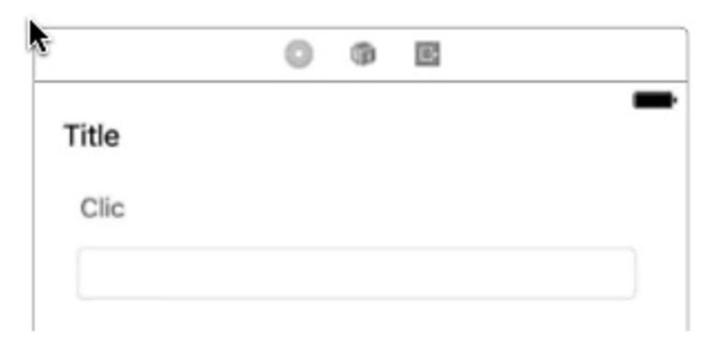


Modifiez la version « *french* », indiquez les paires nom-valeur avec les mêmes clés mais des valeurs en français :

```
"Title"="Appli test langues" ;
"Content"="Cette application est réalisée pour tester les
     ressources texte avec XCode" ;
```

Pour le *storyboard*, il faut le faire une fois que celui-ci est finalisé. Cochez les cases *ad hoc* dans la partie « *Localization* » : XCode crée alors un fichier de chaînes par langue, contenant les étiquettes des différents contrôles. Il ne reste plus qu'à modifier ces fichiers pour les différentes langues choisies.

Exemple avec un *storyboard* simple :



Le fichier de chaînes anglaises :

```
/* Class = "UILabel"; text = "Title"; ObjectID = "6cs-U8-gC5"; */
"6cs-U8-gC5.text" = "Title";

/* Class = "UIButton"; accessibilityLabel = "Clic"; ObjectID = "sTM-I4-fkW"; */
"sTM-I4-fkW.accessibilityLabel" = "Clic";

/* Class = "UIButton"; normalTitle = "Clic"; ObjectID = "sTM-I4-fkW"; */
"sTM-I4-fkW.normalTitle" = "Clic";
```

Le fichier de chaînes françaises :

```
/* Class = "UILabel"; text = "Title"; ObjectID = "6cs-U8-gC5"; */
"6cs-U8-gC5.text" = "Titre";

/* Class = "UIButton"; accessibilityLabel = "Clic"; ObjectID = "sTM-I4-fkW"; */
"sTM-I4-fkW.accessibilityLabel" = "Cliquer ici";

/* Class = "UIButton"; normalTitle = "Clic"; ObjectID = "sTM-I4-fkW"; */
"sTM-I4-fkW.normalTitle" = "Cliquer ici";
```

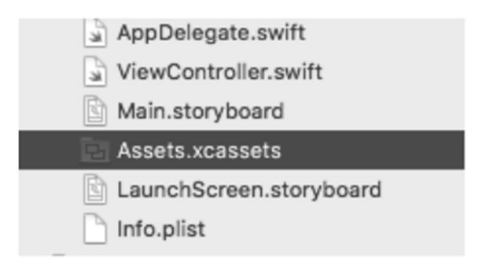
Lors du clic sur le bouton, nous allons tout simplement relire une chaîne localisée par `NSLocalizedString`.



Capture 35 : application XCode localisée, anglais et français

b) Ressources image

Chaque projet XCode contient un fichier `Assets.xcassets` qui sert à référencer l'ensemble des images de l'application, entre autres son icône. Il suffit de sélectionner ce fichier et de glisser une image dans la zone *ad hoc* pour ajouter une image aux ressources. Il y a trois tailles pour chaque image : 1x (standard), 2x et 3x. Vous pouvez fournir les 3 images ou juste une seule (les autres sont redimensionnées dans ce cas).



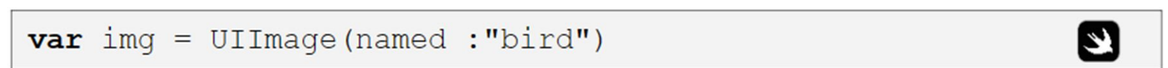




Capture 36 : ImageView avec ressource image XCode

L'utilisation dans le constructeur d'interfaces est simple : dans un contrôle `ImageView`, par exemple, réglez le paramètre `Image` à l'image placée dans `Assets.xcassets`.

L'utilisation dans le code est simple également, on appelle le constructeur de la classe `UIImage` avec le nom de la ressource (ne pas ajouter l'extension), comme dans l'exemple ci-dessous :



```
var img = UIImage(named :"bird")
```

3. Les ressources avec Xamarin.Forms

Les ressources avec *Xamarin* sont intégrées à *l'assembly*[1] .NET et disponibles directement dans l'application. .Net n'oblige pas à utiliser un nom de dossier particulier, vos ressources peuvent être n'importe où dans *l'assembly*. Avec *Xamarin.Forms*, il est recommandé de placer les ressources dans le projet partagé (PCL ou .NETStandard). Un projet de type « *shared* » ne pouvant pas contenir de ressources, il faudrait alors placer les ressources dans chaque projet (iOS, Android, UWP) et on perd l'intérêt de la programmation multiplateforme.

Les ressources peuvent être de plusieurs types :

- Images (PNG, JPG…)
- Fichiers de ressources (RESX)
- Fichiers quelconques

a) Ressources différentes suivant les appareils

Il est possible de localiser les ressources, c'est-à-dire d'avoir des ressources différentes suivant la langue de l'appareil.

---

[1] Un *assembly* .NET est une partie d'une application .NET, en général un fichier .DLL, qui contient du code compilé ainsi que des données incorporées.

Pour cela, on modifie le nom de la ressource en lui ajoutant le code du pays et, éventuellement, de la région. Les codes pays sont normalisés (RFC 4646, ISO 639, ISO 3166). Une liste est visible ici : http://azuliadesigns.com/list-net-culture-country-codes/

Si la ressource est dans un fichier `Textes.resx`, alors :

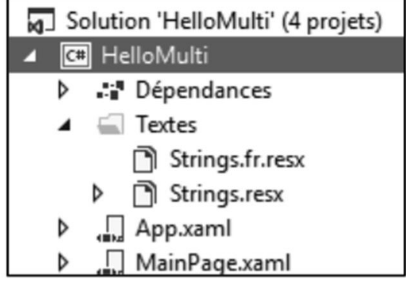
- `Textes.fr.resx` sera utilisé pour une machine en français
- `Textes.en-US` sera utilisé pour une machine en anglais des États-Unis
- `Textes.en-CA` sera utilisé pour une machine en anglais du Canada
- `Textes.resx` sera utilisé pour les autres langues

Il est donc aisé de réaliser avec *Xamarin.Forms* une application multilingue : il suffit que toutes les chaînes de caractères soient placées dans un fichier de ressources, avec un fichier de ressources par langue souhaitée.

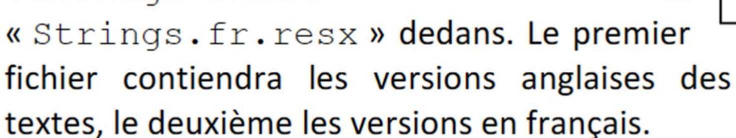
b) Utilisation des ressources textes pour l'internationalisation des applications

Nous allons réaliser une application bilingue français-anglais d'un simple « *hello world* ».

Commençons par démarrer avec Visual Studio 2017 un projet « *cross platform* », avec *Xamarin.Forms* pour l'IHM et un projet portable (PCL ou .NETStandard). Nous allons ensuite créer un dossier « `Textes` » dans le projet portable, et y placer les fichiers de ressource « `Strings.resx` » et « `Strings.fr.resx` » dedans. Le premier fichier contiendra les versions anglaises des textes, le deuxième les versions en français.



Capture 37 : fichiers de ressources Xamarin.Forms



Puis, nous allons éditer la fenêtre principale (`MainPage.xaml`) pour y placer un simple bouton, dans un premier temps sans libellé.

```
<Button HorizontalOptions="Center" VerticalOptions="Center"
    Margin="10" Clicked="Cliquer"/>
```

Nous n'allons pas indiquer « en dur » le texte du bouton, mais dans une ressource. Dans le fichier `Strings.resx`, ajoutez une ressource appelée `TexteBouton` et donnez-lui la valeur « `clic me` ». Ajoutez une ressource du même nom dans `Strings.fr.resx` et donnez-lui la valeur « `clique moi` ».

Nom	Valeur
TexteBouton	clic me !

Nom	Valeur
TexteBouton	Clique moi !

Retournons ensuite dans le fichier XAML de la page : il faut ajouter dans l'entête le chemin des ressources :

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
     xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
     xmlns:local="clr-namespace:HelloMulti"
     x:Class="HelloMulti.MainPage"
     xmlns:resx="clr-namespace:HelloMulti.Textes"
>
```

La ligne ajoutée (ici en gras) indique simplement que le nom `resx` fait référence au dossier `Textes` du projet `HelloMulti` (bien sûr il faut l'adapter à vos propres noms).

Nous pouvons ensuite indiquer que le texte du bouton est lié à la ressource `TexteBouton` du fichier `Strings` :

```
<Button Text="{x:Static resx:Strings.TexteBouton}" />
```

Le nom de la ressource est donc tout simplement composé de celui du fichier (ici `Strings`) et du nom de la ressource.

L'exécution de l'application donne le résultat suivant (test avec un Windows en français, un Android et un iOS en anglais) :



Capture 38 : Illustration des ressources textes avec Xamarin.Forms (Windows, Android, iOS)

Passons à présent au code associé au clic sur ce bouton : nous souhaitons faire afficher un message. Ce message doit également être dans les

ressources. Ajoutons donc dans nos fichiers `resx` une ressource appelée « `Message` » et contenant le message de la ressource, ainsi qu'une ressource appelée « `Titre` » :

Nom	Valeur
Message	Salut le monde !
TexteBouton	Clique moi !
Titre	Message

Nom	Valeur
Message	Hello, world !
TexteBouton	clic me !
Titre	Message

La fonction de réponse au clic du bouton fera ainsi référence à cette ressource :

```
private void Cliquer(object sender, EventArgs args)
{
 DisplayAlert(Textes.Strings.Titre, Textes.Strings.Message,
     "Ok");
}
```

Le résultat dépend de la plateforme et de sa langue (ici, avec Windows en français et Android/iOS en anglais) :



c) Utilisation des ressources images

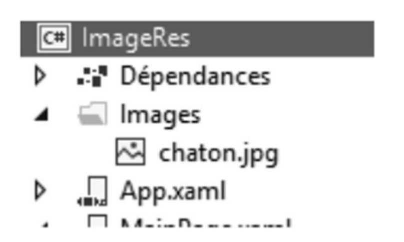
Les images sont des ressources très pratiques dans nos applications : l'icône de l'application, l'écran de démarrage (*splash screen*), diverses images utilisées en fond d'écran… Plutôt que d'utiliser des fichiers livrés avec l'application, il est plus simple d'embarquer l'image dans les ressources.

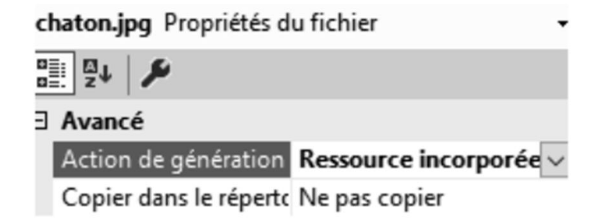
Si les images sont placées dans les pages XAML partagées, il est nécessaire que les ressources *ad hoc* soient dans le projet commun (PCL ou .NETStandard). En revanche, si l'image est directement liée à l'application (icône, *splash screen*…) il faut la mettre dans le projet correspondant au système, en respectant la norme en vigueur (voir VI.1.c) pour Android et

VI.2.b) pour iOS). Nous traiterons dans cette partie des images partagées qui vont se retrouver dans les pages de l'application.

Le contrôle *Xamarin.Forms* utilisé pour afficher une image est tout simplement… `Image`.

Pour utiliser une ressource image, il suffit de placer un fichier image dans le projet (par exemple dans un dossier `Images`) et régler sa propriété de génération à « ressource incorporée » :





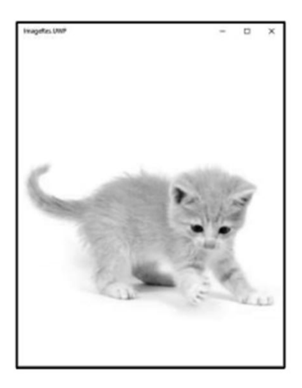
La balise XAML pour l'image est la suivante, avec au minimum comme propriétés son nom, pour l'identifier, et son aspect, redimensionné ou non :

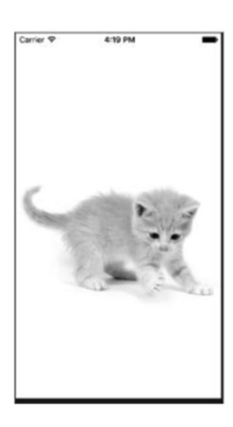
```xml
<Image x:Name="image" Aspect="AspectFit"  />
```

Pour associer l'image avec la ressource il est nécessaire d'utiliser du code C#, par exemple dans le constructeur de la page en utilisant le nom composé du nom du projet, du nom du dossier et du nom du fichier (adaptez suivant le cas). L'indication de *l'assembly* source est préférable (obligatoire pour Windows) :

```csharp
var assembly = typeof(MainPage).GetTypeInfo().Assembly;
image.Source =
    ImageSource.FromResource("ImageRes.Images.chaton.jpg",
    assembly);
```

Le résultat est quasi identique suivant la plateforme :





Capture 39: intégration d'images ressources avec Xamarin.Forms (Windows, Android, iOS)

Un cas particulier : l'image de fond utilisée dans la page de l'application. Chaque `Page` possède une propriété `BackgroundImage` qui est une chaîne et doit contenir le nom de l'image (par exemple `fond.jpg`). Cette image ne peut hélas pas être placée n'importe où :

- Projet Android : dans `Resources/drawable` avec comme action de génération `AndroidResource`
- Projet iOS : dans `Resources`, avec comme action de génération `BundleResource`
- Projet UWP : à la racine du projet, avec comme action de génération `Contenu`

Attention, si la page est une page à onglets (`TabbedPage`), ce sont les différentes pages onglets qui doivent avoir une image de fond.

Il est également possible d'utiliser une image des ressources sans passer par du code C#, mais dans ce cas, l'image doit être placée comme ressource dans chaque projet, comme pour l'image de fond décrite plus haut.

4. Les ressources avec Qt

Qt produit un exécutable qui peut incorporer un certain nombre de ressources, textes, images, etc… Toutes ces ressources peuvent être vues comme des fichiers et les classes Qt les utilisant (`QIcon` par exemple) peuvent y accéder simplement.

Pour ajouter des fichiers (type quelconque) en tant que ressources, il faut créer un fichier de type « collection de ressources » (`qrc`). Dans QtCreator, faire *Ajouter nouveau → Qt → Qt Resource File*.

QtCreator comprend un éditeur de collection de ressources. Dans cet éditeur, il suffit d'indiquer un préfixe (pour regrouper les ressources) puis le(s) fichier(s) à inclure.

Une fois les ressources compilées, il est aisé d'y accéder dans l'application, le chemin est le suivant :

```
:/prefixe/nom_ressource
```

Par exemple, un fichier `splash.jpg` situé dans une ressource au préfixe `/images`, sera accessible avec l'URL suivante :

```
:/images/splash.jpg
```

Il est même possible, dans le fichier QRC, d'indiquer une langue pour une ressource : cette ressource ne sera utilisée que pour cette langue. Cela permet, par exemple, des fichiers sonores ou images différents en français et en anglais.

a) Utilisation des ressources pour l'internationalisation

Qt fournit un mécanisme d'internationalisation assez complexe mais complet. Il faut indiquer dans le code (côté IHM) les chaînes devant être traduites par la macro `tr`. Les chaînes de l'interface utilisateur sont obligatoirement traductibles.

Il faut choisir un nom commun pour les chaînes (par exemple `textes`) et faire une version du fichier par langue (exemple : `textes_fr_FR`). Ne créez pas les fichiers directement, mais référencez-les dans le fichier projet :

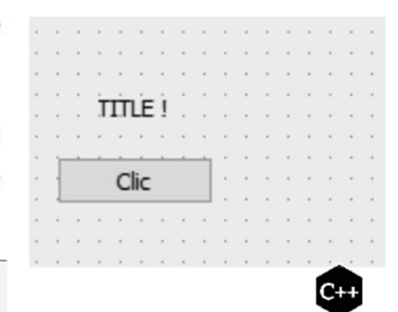
```
TRANSLATIONS = textes_fr_FR.ts
```

Durant le développement de l'application, utilisez la macro `tr` pour indiquer les chaînes à traduire.

Par exemple, l'appel d'une boite de message :

```
QMessageBox::information(this,tr("Title"),tr("This is a
    message"));
```

Dans l'IHM, indiquez simplement la chaîne dans la langue par défaut (anglais ici) :



Dans le fichier principal de l'application (`main.cpp`), n'oubliez pas d'inclure le fichier de la classe de traduction :

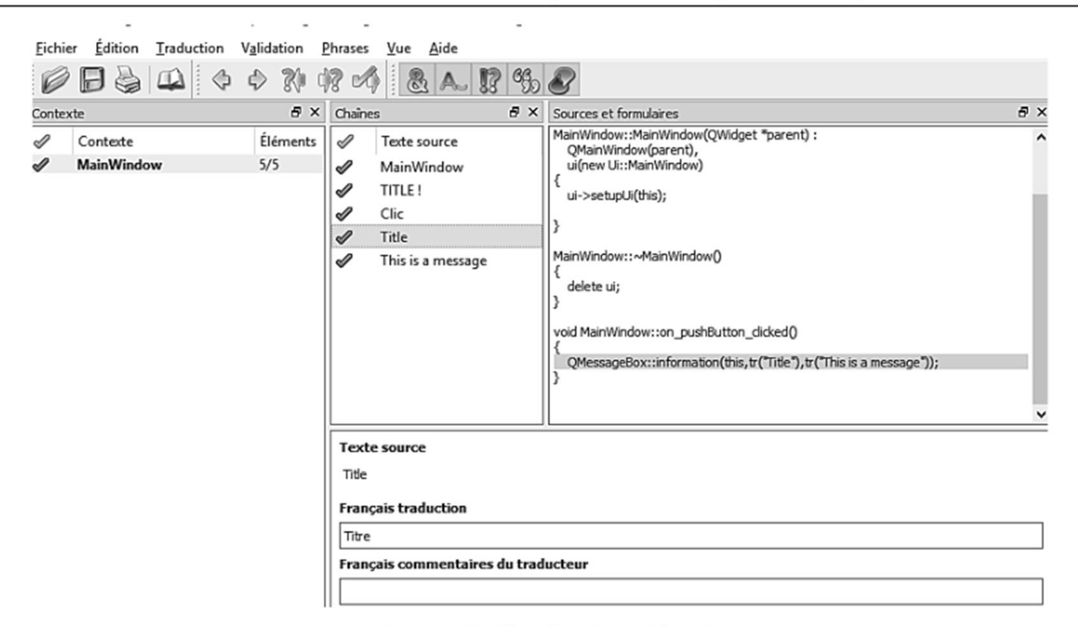
```
#include <QTranslator>
```

Il faut ensuite utiliser l'utilitaire `QtLinguist` livré avec Qt, composé de trois exécutables :

- `Lupdate` crée ou met à jour le(s) fichier(s) de traductions indiqué(s) dans le projet à partir des chaînes détectées dans l'application
- `Linguist` permet d'aider à la traduction des chaînes
- `Lrelease` permet de publier les traductions dans l'application (une sorte de compilation)

`QtLinguist` est normalement placé dans le dossier `bin` de *MinGW*, sinon faites une recherche sur l'application `linguist`.

Il faut commencer par créer le fichier de traduction à l'aide de `lupdate`. Vous pouvez utiliser la ligne de commande, ou le menu « *outil* → *externe* → *linguist* → *lupdate* ». Un fichier portant le nom indiqué dans le fichier projet est créé. Vous pouvez l'éditer à la main (c'est du XML simple) ou en utilisant `Linguist`, pour donner la traduction des chaînes de votre application.



Capture 40 : l'application QtLinguist

Une fois le fichier traduit, il suffit de le publier, soit directement dans `linguist` (*fichier → publier*) soit dans QtCreator (*outils → externe → linguist → lrelease*). Un fichier avec l'extension `.qm` est généré (par exemple `textes_fr_FR.qm`). Ce fichier devant être accessible à l'application, il doit être placé dans les ressources. Créez donc un fichier QRC dans le projet, et ajoutez-y le(s) fichier(s) `qm` de traductions :



Il faut ensuite créer, dans la fonction `main`, avant le code nécessaire pour lancer la fenêtre principale mais après celui créant l'application, une instance de `QTranslator`, la faire charger le bon fichier de langue (suivant la langue du système) puis l'installer dans l'application (remarquez le chemin particulier du fichier, situé dans une ressource) :

```
QApplication a(argc, argv);
QTranslator trans;
QString file = ":/strings/textes_"+QLocale::system().name();
bool ok = trans.load(file);
a.installTranslator(&trans);

MainWindow w;
w.show();
```

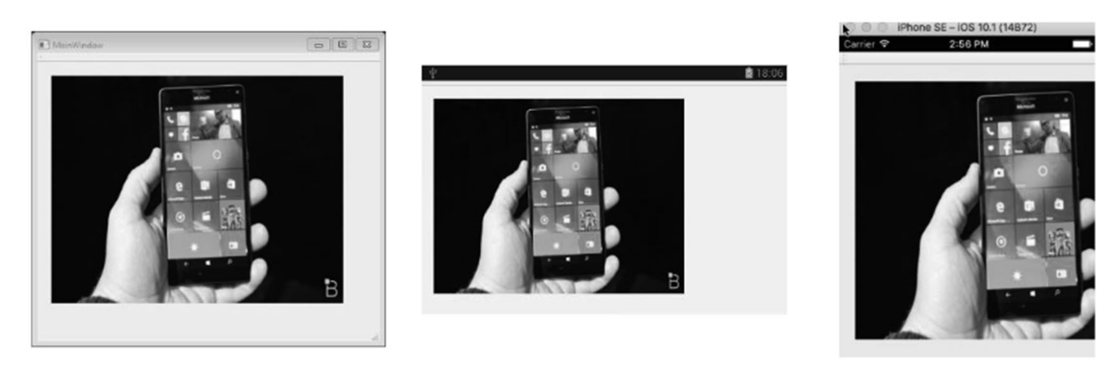


Capture 41 : application multilingues avec Qt sous Windows (français), iOS (français), Android (anglais)

b) Utilisation des ressources images

Les images ne sont qu'un type comme un autre de fichier, et l'accès via les ressources est assez simple.

Pour afficher une image, il faut utiliser un composant `QLabel` (oui, cela peut paraître étrange) et affecter sa propriété `Pixmap` à la ressource désirée, tout simplement.



Capture 42 : ressources images avec Qt : windows, android, iOS

5. Les ressources avec Cordova

a) Ressources texte

Comme Cordova ne crée pas réellement d'exécutable mais un site web, il n'y a pas à proprement parler de ressources texte incorporées dans le code ! Il est néanmoins possible d'obtenir une application accessible en plusieurs langues en fonction de la langue du système.

Plusieurs possibilités coexistent :

- L'utilisation du *plugin* `cordova-plugin-globalization` (10)
- L'utilisation de la bibliothèque `JQuery.localize` (11)
- Les API *ECMAScript* (12)

La bibliothèque `JQuery.localize` est assez simple à utiliser. Il faut bien entendu ajouter les deux bibliothèques (`JQuery` et `JQuery.localize`) dans votre projet Cordova (dans le dossier `www/scripts` par exemple) et les intégrer dans votre fichier `index.html` :

```
<script src="scripts/jquery-3.3.1.min.js"></script>
<script src="scripts/jquery-localize.js"></script>
```

Les fichiers de ressources sont des fichiers JSON[1] simples qui contiennent autant de paires *nom-valeur* que de chaînes à stocker. Les fichiers doivent avoir le même nom (par exemple `strings`) avec un suffixe dépendant du code de langue (`-fr` pour le français, `-en` pour l'anglais, etc…).

Si nous avons, par exemple, deux chaînes : une appelée « `title` » et une appelée « `text` », nous aurons les fichiers JSON suivants pour le français et l'anglais :

Strings-fr.json	{JSON} Strings-en.json
`{ "title": "Bonjour !", "text": "Ceci est un texte en français" }`	`{ "title": "Hello !", "text": "This is an english text" }`

Dans le fichier `index.js`, déclarez une variable globale pour stocker ces chaînes :

```
var strings ;
```

Dans la fonction `onDeviceReady`, (une fois le périphérique prêt) il suffit d'appeler le code suivant pour charger les fichiers de langue (en précisant le nom sans suffixe du fichier JSON à utiliser) :

[1] JavaScript Object Notation : représentation d'un objet sous forme d'une chaîne de caractères.

```
$("[data-localize]").localize("strings", {
            callback: function (data, defaut) {
                strings = data;
                defaut(data);
            }
        });
```

Lorsque cette fonction est lancée, elle charge les JSON dans l'objet `strings` (qui seront donc utilisable dans le code du script) et parcourt le document HTML pour placer les bons textes.

Dans le fichier HTML, si l'on veut faire référence à une chaîne ressource dans une balise, on utilise un attribut `data-localize`, ici avec un titre :

```
<h1 data-localize="title">Title</h1>
```

Dans un code *Javascript*, si l'on veut faire référence à une chaîne ressource, on accède à une propriété de l'objet `strings` chargé plus haut :

```
alert(strings.text);
```

Exemple d'une application utilisant les ressources texte exécutée dans un navigateur :





Capture 43 : ressources textes avec Cordova, en français et en anglais

Il peut arriver que les ressources texte ne soient pas utilisées pour la traduction mais comme source de données. Dans ce cas, la ressource est un fichier stocké sur le serveur (Javascript est une API pour un navigateur). Pour

y accéder, il faut donc utiliser une requête Ajax[1] (un « `get` ») et récupérer la réponse (du texte brut, du *json*…).

Exemple de code Javascript (ECMAScript 6, utilisant les promesses[2] pour simplifier la gestion de l'asynchronisme) qui lit un fichier texte (`word.txt`, placé dans le dossier `res` du dossier `www`) et l'affiche dans un élément HTML de la page (un paragraphe dont l'identifiant est texte) :

```
fetch("/res/words.txt").then(function (reponse) {
     reponse.text().then(function (txt) {
        document.getElementById("texte").innerHTML = txt;
     });
}).catch(function (x) {
     alert(x);
});
```

b) Ressources image

Pour les ressources images, c'est encore plus simple puisqu'en HTML, une image utilise systématiquement un fichier extérieur ! La balise `IMG` contient donc directement le chemin vers la ressource, à savoir le fichier image utilisé pour l'affichage.

[1] *Asynchronous Javascript And Xml* : ensemble de technologies permettant le dialogue entre le client http (navigateur) et le serveur sans recharger la page web en cours.

[2] Les promesses sont une nouveauté ECMAScript 6 pour faciliter la gestion des opérations asynchrones tout en gardant une lisibilité correcte du code Javascript.

VII.La persistance des données

Toutes les données manipulées par une application (mobile ou autre) sont présentes dans la mémoire de l'appareil. Lorsque l'application se ferme (fin normale, erreur, extinction de l'appareil…), celles-ci disparaissent. Il est important que certaines données persistent entre deux sessions : les paramètres de l'application (choix des couleurs par exemple), certaines données manipulées, etc.

Il existe plusieurs méthodes pour avoir des données persistantes dans nos applications mobiles. Il est important d'assurer une sécurité minimale pour les données persistantes. Les systèmes mobiles sont (relativement) sécurisés sur ce principe : il ne faut pas qu'une application puisse manipuler les données d'une autre application, par exemple. Néanmoins, il est parfois nécessaire à une application d'utiliser des données partagées. Prenons l'exemple des photos de votre téléphone : l'application gérant l'appareil photo doit pouvoir les écrire, une application permettant de les afficher doit pouvoir les lire, une application permettant de modifier une photo (supprimer les yeux rouges, par exemple) de les lire et de les écrire.

Les systèmes mobiles découpent l'espace de stockage en plusieurs parties :

- L'espace réservé à l'application : inaccessible aux autres applications et accessible directement en lecture/écriture à l'application elle-même.
- L'espace partagé réservé aux images (photos, etc…) : accessible à toute application (lecture et écriture) qui indique dans son manifeste le souhait d'utiliser ce dossier.
- L'espace partagé réservé aux musiques : même principe que pour les images.
- L'espace partagé réservé aux vidéos (parfois confondu avec celui des images) : même principe.
- L'espace partagé réservé aux documents : même principe.
- L'espace réservé au système : accessible au système uniquement.
- Le reste de l'espace : accessible uniquement sur interaction visuelle avec l'utilisateur.

Il est intéressant de constater que les trois systèmes actuels (Android, Windows, iOS) sont similaires sur ces principes.

1. Paramètres de l'application

Tous les systèmes mobiles intègrent la notion de paramètres d'une application : chaque application peut gérer un petit nombre de données directement dans son espace privé avec un accès simple. Ce mode est à privilégier pour la persistance des données paramétrant l'application ou les données de celle-ci si elles sont peu nombreuses.

a) Paramètres Android

Sous Android, les paramètres sont des paires clé/valeur où la clé est une chaîne de caractères identifiant le paramètre, et la valeur la donnée à stocker. Les types stockables sont :

- Types primitifs (`int`, `boolean`, `float`, `long`)
- Chaîne de caractères (`String`)
- Ensemble de chaînes (`Set<String>`)

Tout type pouvant être transformé en chaîne (*sérialisable*) est donc possible.

Les paramètres sont appelés **préférences** (`Preferences`) sous Android. La classe `Activity` fournit les fonctions `getPreferences` et `getSharedPreferences` pour accéder aux paramètres. Dans les deux cas, l'objet renvoyé est un `SharedPreferences`. La seule différence est que `getSharedPreferences` permet de spécifier un nom pour avoir différents jeux de paramètres.

Pour utiliser les paramètres de l'activité en cours :

```
SharedPreferences prefs = getPreferences(MODE_PRIVATE) ;
```

Pour utiliser un jeu de paramètres particulier (identifié par une chaîne) :

```
SharedPreferences prefs =
    getSharedPreferences("mesprefs",MODE_PRIVATE) ;
```

Pour lire une valeur, il suffit d'utiliser une fonction `getXXX` fournie par l'objet `SharedPreferences` : `getInt` pour un entier, `getString` pour une chaîne, etc… Les fonctions demandent en paramètre une chaîne

de caractères qui identifie (clé) la valeur souhaitée et la valeur par défaut (si la valeur n'est pas trouvée). Par exemple, le code suivant lit les paramètres « `nom` » et « `age` » :

```java
String nom = prefs.getString("nom","") ;
int age = prefs.getInt("age",0) ;
```

Pour écrire une valeur dans les paramètres, il faut utiliser une autre classe, interne à `SharedPreferences` : `Editor`. Un objet `Editor` (éditeur) possède des opérations `putXXX` (`putString` pour une chaîne, `putInt` pour un entier…) qui permettent d'associer à une clé une valeur. La sauvegarde ne se fait pas directement : il faut appeler la méthode `apply` de l'objet `Editor` pour qu'elle soit effective. L'exemple suivant fixe les paramètres « `nom` » et « `age` » :

```java
SharedPreferences.Editor ed = prefs.edit() ;
ed.putString("nom","toto") ;
ed.putInt("age",15) ;
ed.apply() ;
```

b) Paramètres Windows

Sous la plateforme UWP, et donc pour les versions mobiles de Windows, il existe deux magasins de paramètres :

- Un magasin local à l'application, stocké sur l'appareil
- Un magasin « *cloud* », stocké dans le dossier *Onedrive* de l'utilisateur et utilisable par tout appareil utilisant le même compte (si l'appareil est relié à Internet et utilise un compte Microsoft)

Le fonctionnement des magasins est identique et utilise un objet de type `ApplicationDataContainer`.

Pour accéder au magasin local de l'application :

```csharp
ApplicationDataContainer prefs =
    ApplicationData.Current.LocalSettings ;
```

Pour accéder au magasin « *cloud* » de l'application :

```csharp
ApplicationDataContainer prefs =
    ApplicationData.Current.RoamingsSettings ;
```

Le magasin fournit un tableau associatif où la clé (identifiant) est une chaîne et la valeur un `Object` sérialisable.

L'exemple suivant lit les paramètres « `nom` » et « `age` » :

```
String nom = prefs.Values["nom"] ;
int age = Convert.ToInt32(prefs.Values["age"]) ;
```

L'écriture des paramètres est similaire à la lecture :

```
prefs.Values["nom"] = "toto" ;
prefs.Values["age"] = 15.ToString() ;
```

c) Paramètres iOS

iOS fournit un système pour simplement sauvegarder les données de l'application : les « valeurs par défauts » (*User defaults*).

Les types utilisables sont `String`, `Number`, `Date`, `Bool`, `Array`, `Dictionnary` et `Data`, ce qui permet de sauvegarder à peu près tout ce qu'on veut.

Les données sont sauvegardées dans un fichier de type « listes de propriétés » (fichiers `plist`) propre à l'application, avec un système de clé-valeur : les clés sont obligatoirement des chaines de caractères.

Il faut utiliser la classe `UserDefaults` :

```
let userDef = UserDefaults.standard
```

L'exemple suivant est utilisé pour lire un paramètre « `nom` » et un paramètre « `age` » :

```
let nom = userDef.string("nom")
let age = userDef.integer("age")
```

L'exemple suivant montre comment écrire ces valeurs. Les valeurs ne sont réellement écrites que lors de l'appel à la fonction `synchronize` :

```
userDef.set("toto",forKey :"nom")
userDef.set(15,forKey :"age")
userDef.synchronize()
```

Swift

d) Paramètres Xamarin.Forms

Xamarin.Forms ne fournit pas nativement de solution pour gérer les paramètres de l'application de manière portable.

Heureusement, il existe un *plugin* Xam.Plugins.Settings (13) qui permet d'accéder aux paramètres avec un projet partagé .NETStandard (donc utilisable avec Android, iOS, Windows).

L'installation du *plugin* doit se faire dans **tous** les projets de la solution (partagé, Android, iOS, Windows UWP). Il suffit pour cela d'exécuter une commande *NuGet* comme la suivante (la version à l'heure où j'écris ces lignes est la 3.1.1 mais il vaut mieux installer la version la plus récente) :

```
Install-Package Xam.Plugins.Settings -Version 3.1.1
```

Nous n'avons besoin que d'une seule classe (CrossSettings) qui possède deux opérations, une pour lire (GetValueOrDefault) et une pour écrire (AddOrUpdateValue). Les types possibles sont Int, Int64, Double, String, DateTime, Bool, Float, Decimal, Guid. Tout type capable de se sérialiser (donc transformable en chaîne) est également possible.

L'exemple suivant permet d'écrire les paramètres « nom » et « age » :

```
CrossSettings.Current.AddOrUpdateValue("nom","toto") ;
CrossSettings.Current.AddOrUpdateValue("age",15) ;
```

L'exemple suivant permet de lire les paramètres « nom » et « age » :

```
String nom =
     CrossSettings.Current.GetValueOrDefault("nom","") ;
int age = CrossSettings.Current.GetValueOrDefault("age",0) ;
```

e) Paramètres Cordova

HTML5 définit la possibilité pour un navigateur d'utiliser un stockage local simplement accessible par des paires clé/valeur. Ce stockage est limité à des chaînes (donc tout objet *sérialisable*) et à des petites quantités de données (<5Mo).

L'accès à l'objet Javascript gérant le stockage local se fait par l'objet window :

```
var storage = window.localStorage ;
```

L'exemple suivant stocke un nom et un âge :

```
storage.setItem("nom","toto") ;
storage.setItem("age","15") ;
```

L'exemple suivant lit un nom et un âge depuis le stockage local :

```
var nom = storage.getItem("nom") ;
var age = storage.getItem("age") ;
```

f) Paramètres Qt

Le *framework* Qt contient la classe `QSettings` (14), dont le but est de gérer les paramètres de manière portable.

Un objet `QSetting` a besoin lors de sa construction de deux chaînes qui permettent de spécifier l'organisation et l'application. Si ces chaînes sont omises, `QSettings` utilise celle du manifeste.

L'exemple suivant indique une organisation et une application spécifique :

```
QSettings prefs("AG","TestQSettings") ;
```

La classe utilise le type `QVariant` pour le stockage, et peut donc stocker des chaînes, entiers, etc… Elle est plus générale qu'une simple lecture/écriture de paramètres mais nous ne rentrerons pas dans le détail ici.

Le code suivant écrit des paramètres comprenant un nom et un âge :

```
prefs.setValue("nom","toto") ;
prefs.setValue("age",15) ;
```

Le code suivant lit ses mêmes paramètres :

```
QString nom = prefs.value("nom").toString() ;
int age = prefs.value("age").toInt() ;
```

2. Fichiers

Sur l'ensemble des systèmes mobiles, une application ne peut accéder à l'intégralité des fichiers, pour des raisons de sécurité. La gestion de ceux-ci

est donc bien moins souple qu'avec une programmation « classique » où il suffit de donner le chemin du fichier pour pouvoir y accéder.

Chaque application possède un dossier privé qui lui est associé dans lequel elle peut manipuler, en lecture/écriture, des fichiers de type quelconque. Ce dossier n'étant pas accessible aux autres applications, il est protégé contre toute intrusion, suppression, virus, etc.

Il existe également sur le téléphone des espaces « partagés » où toutes les applications peuvent lire et écrire des fichiers, sous réserve d'en avoir demandé l'autorisation :

- Les documents
- Les images
- Les musiques/sons
- Les vidéos
- Le support externe (carte SD par exemple)

a) Android

Chaque activité possède des opérations pour accéder aux fichiers privés de l'application. En Java, les fichiers sont manipulés grâce à des flux (`FileOutputStream` / `FileInputStream`) qui doivent obligatoirement être fermés correctement.

Les opérations les plus courantes sur les fichiers sont :

- `fileList` : liste les différents fichiers privés
- `openFileOutput` : ouvre (ou crée) un fichier en écriture
- `openFileInput` : ouvre un fichier en lecture

Pour lire un fichier, par exemple, il faut commencer par ouvrir un flux en lecture :

```
FileInputStream flux = openFileInput("nomfichier") ;
```

Un fois ce flux ouvert, il existe plusieurs méthodes pour lire les données.

Un objet de type `InputStreamReader` fournit des opérations de bas niveau pour lire un certain nombre d'octets depuis le flux.

```
InputStreamReader isr = new InputStreamReader(flux) ;
if(isr.ready())
{
```

Java

```
    int c = isr.read() ; // 1 seul caractère
}
```

Un objet de type `BufferedReader` fournit, outre des tampons mémoire, des méthodes de plus haut niveau pour lire des chaînes, des nombres, etc. à partir d'un `InputStreamReader`.

```
BufferedReader br = new BufferedReader(new
    InputStreamReader(flux)) ;
```

Il est également possible de directement récupérer un objet Java (sérialisation) à partir d'un fichier :

```
ObjectInputStream ois = new ObjectInputStream(flux) ;
LaClasse obj = (LaClasse)ois.readObject() ;
```

Attention : dans tous les cas, il faut absolument refermer le flux ouvert, en tenant compte des nombreuses possibilités d'exception ! L'idiome est le suivant :

```
FileInputStream flux = openFileInput(nomFichier) ;
try {
// on travaille avec le flux
}
finally{
    flux.close() ;
}
```

Il est bien entendu possible d'ajouter une clause `catch` si l'on veut capturer l'exception, mais la présence de la clause `finally` garantit que le flux sera refermé, quoi qu'il arrive.

En écriture, le principe est similaire. Pour ouvrir (ou créer) un fichier :

```
FileOutputStream flux = openFileOutput(nomFichier) ;
```

Pour écrire des données en « bas niveau », l'objet `OuputStreamWriter` fournit les opérations `write` pour écrire un (ou plusieurs) octet(s) :

```
OuputStreamWriter ows = new OutputStreamWriter(flux) ;
ows.write(0xFF) ;
```

Pour écrire avec des opérations de plus haut niveau, la classe `BufferedWriter` présente des opérations pour écrire des chaînes, nombres, etc…

```
BufferedWriter bw = new BufferedWriter(new
     OutputStreamWriter(flux)) ;
bw.write("du texte…") ;
```

Et bien sûr, il est possible de sérialiser des objets directement…

```
ObjectOutputStream oos = new ObjectOutputStream(flux) ;
oos.writeObject(obj) ;
```

b) Windows

Chaque application possède un dossier privé, accessible en lecture et écriture, pour stocker ses données.

Le programme s'exécutant dans un environnement de type « bac à sable »[1], l'application ne peut pas accéder à n'importe quel fichier du système. Les dossiers accessibles sont :

- Le dossier d'installation de l'application, en lecture seulement : `ApplicationModel.Package.Current.InstalledLocation`
- Le dossier local des données de l'application : `ApplicationData.Current.LocalFolder`
- Le dossier itinérant des données de l'application (stocké dans le *onedrive* de l'utilisateur) : `ApplicationData.Current.RoamingFolder`
- Les dossiers « spéciaux », ou « connus », qui sont accessibles sous réserve que l'application en demande l'autorisation (`KnowFolders`) : documents, images, vidéos, musique, etc…
- Tout dossier auquel on accède via une boite de dialogue standard du système (`Picker`) dans la même opération : cela garantit que l'utilisateur voit bien et confirme l'accès à un fichier/dossier.

[1] *Sandbox* : espace sécurisé, isolé du reste du système. L'application ne peut ainsi pas modifier le système ou les autres applications.

Les types utilisés sont `StorageFolder` pour les dossiers et `StorageFile` pour les fichiers. Pratiquement toutes les opérations de ces classes sont asynchrones.

Les fichiers (`StorageFile`) peuvent être accédés via les opérations de haut niveau (texte) de la classe `FileIO` ou via les flux (`Stream`) en lecture et/ou écriture.

Exemple pour créer un fichier texte dans le dossier privé de l'application :

```
StorageFolder folder = ApplicationData.Current.LocalFolder ;
StorageFile file = await folder.CreateFileAsync("test.txt") ;
await FileIO.WriteTextAsync(file,"Ceci est du texte à
     écrire…") ;
```

Remarquez que les appels utilisent le mot-clé `await` et que donc ces lignes doivent être dans une fonction qui est `async` (voir le chapitre X.5.a) sur la programmation asynchrone).

Exemple pour lire un fichier image récupéré par une boite de dialogue utilisateur (`Picker`) :

```
FileOpenPicker picker = new FileOpenPicker() ;
picker.SuggestedStartLocation =
     PickerLocationId.PicturesLibrary ;
picker.FileTypeFilter.Add(".png") ;
StorageFile file = await picker.PickSingleFileAsync() ;

if(file !=null)
{
     using(var flux = await
     file.OpenAsync(FileAccessMode.Read))
     {
        BitmapImage bmp = new BitmapImage() ;
        await bmp.SetSourceAsync(flux) ;
        // on fait ce qu'on veut de l'image…
     }
}
```

c) iOS

iOS fournit des dossiers spécifiques pour les documents, la musique, les images, la configuration de l'application. Il n'est pas possible d'accéder à tout dossier du système de fichiers.

La classe `FileManager` permet d'obtenir les dossiers suivants :

- `applicationDirectory` : la racine des applications
- `libraryDirectory` : les documents divers visibles par l'utilisateur
- `documentDirectory` : les documents généraux
- `moviesDirectory` : les vidéos
- `picturesDirectory` : les images
- `sharedPublicDirectory` : répertoire commun à toutes les applications

Pour obtenir le dossier, il faut appeler la méthode `urls` du `FileManager` :

```
let images = FileManager.default.urls(for :
     .picturesDirectory, in : .userDomainMask).first
```

L'exemple suivant écrit un fichier texte dans le dossier des documents (le texte est issu d'une zone de saisie) :

```
if let docs =
     FileManager.default.urls(for:.documentDirectory,
     in:.userDomainMask).first
{
     let file = docs.appendingPathComponent("test.txt")
     do{
        try saisie.text?.write(to:file, atomically:false,
     encoding:.utf8)
     }
     catch {
     }
}
```

La lecture d'un fichier texte depuis le dossier des documents est tout aussi simple (ici le fichier est recopié dans la zone de saisie) :

```
if let docs =
     FileManager.default.urls(for:.documentDirectory,in:.user
     DomainMask).first{
     do{
        let file = docs.appendingPathComponent("test.txt")
        try saisie.text = String(contentsOf : file)
     }
     catch{
     }
}
```

d) Xamarin

Chaque plateforme ayant un système de fichiers spécifique et une méthode différente pour accéder aux fichiers, *Xamarin.Forms* ne propose aucune approche multiplateforme pour ça.

Comme souvent, il existe des *plugins* qui permettent d'accéder aux fichiers de manière multiplateforme (15).

Le *plugin* `Xamarin.Plugin.FilePicker` permet d'ouvrir n'importe quel fichier du système car il utilise une boite de dialogue de fichier. Il fournit directement le contenu du fichier sous forme d'un tableau d'octets. L'exemple suivant fait choisir un fichier à l'utilisateur et le charge, sous forme de texte UTF8, dans un *label* nommé `txt` :

```
FileData data = await CrossFilePicker.Current.PickFile();
if(data!=null) {
     string contenu =
     Encoding.UTF8.GetString(data.DataArray);
     txt.Text = contenu;
}
```

Le *plugin* `PCLStorage` permet, quant à lui, d'utiliser les fichiers privés de l'application (*Local* et *Roaming*). L'exemple suivant montre comment utiliser ce *plugin* pour écrire un fichier texte dans le dossier local de l'application (le texte est issu d'un éditeur) :

```
IFolder local = FileSystem.Current.LocalStorage;
IFile file = await local.CreateFileAsync("test.txt",
     CreationCollisionOption.ReplaceExisting);
await file.WriteAllTextAsync(editor.Text);
await DisplayAlert("Sauve", "Ecriture fichier terminée",
     "ok");
```

Voici comment lire un fichier texte sauvegardé dans le dossier local de l'application :

```
IFolder local = FileSystem.Current.LocalStorage;
IFile file = await local.GetFileAsync("test.txt");
txt.Text = await file.ReadAllTextAsync();
```

e) Qt

Les API de Qt pour la gestion des fichiers (`QFile`, `QDir`…) peuvent être utilisées dans les environnements « bac à sable » des applications mobiles,

sous certaines conditions. Pour la lecture et/ou l'écriture, il est possible d'utiliser des flux (haut niveau).

Pour l'instant, seuls les dossiers privés de l'application sont utilisables par Qt sur Windows, iOS et Android. Les dossiers de sélection de fichier (`QFileDialog`) ne fonctionneront pas (en tout cas pas de manière portable sur les 3 plateformes étudiées).

Le code suivant permet de sauvegarder un fichier texte (le texte étant issu d'une zone de saisie) :

```
QStringList liste= QStandardPaths::standardLocations(
     QStandardPaths::DataLocation);
if(liste.count()>0)
{
     QString path = liste[0];
     QString fileName = path+QDir::separator()+"test.txt";
     QFile file(fileName);
     if(file.open(QIODevice::WriteOnly))
     {
        QTextStream flux(&file);
        flux << ui->editor->text();
     }
}
```

Voici maintenant commet charger un fichier texte situé dans le dossier local de l'application (le texte étant chargé puis transféré dans un label) :

```
QString fileName =
     QStandardPaths::locate(QStandardPaths::AppLocalDataLocat
     ion, "test.txt");
if(!fileName.isEmpty()) {
     QFile file(fileName);
     if(file.open(QIODevice::ReadOnly))   {
        QTextStream flux(&file);
        QString txt = flux.readAll();
        ui->txt->setText(txt);
     }
}
```

f) Cordova

L'accès aux fichiers est possible en utilisant le *plugin* standard `cordova-plugin-file`.

Le *plugin* permet d'accéder aux emplacements suivants :

- Installation de l'application, en lecture seule : `cordova.file.applicationDirectory`
- Dossier local privé de l'application, en lecture : `cordova.file.applicationStorageDirectory`
- Dossier local des données de l'application, en lecture/écriture : `cordova.file.dataDirectory`
- Dossier distant des données de l'application (iOS/Windows) : `cordova.file.syncedDataDirectory`

Avant de faire toute opération sur un fichier, il faut appeler l'opération `requestFileSystem` qui fournit à sa fonction de rappel un objet `fileSystem` qui peut être utilisé :

```
window.requestFileSystem(LocalFileSystem.PERSISTENT,0,
    function(fs){
// utilise le fileSystem fs
}) ;
```

L'objet `root` du `fileSystem` permet d'accéder au dossier lui-même.

Par exemple, pour sauver un texte dans un fichier du dossier de l'application :

```
var texte = document.getElementById("editor").value;
window.requestFileSystem(LocalFileSystem.PERSISTENT, 0,
   function (fs) {
     fs.root.getFile("test.txt", { create: true }, function
     (file) {
        file.createWriter(function (writer) {
        writer.write(texte);
        });
   });
});
```

Pour lire un fichier texte du dossier de l'application et le transférer dans un paragraphe du document :

```
window.requestFileSystem(LocalFileSystem.PERSISTENT, 0,
 function (fs)
 {
     fs.root.getFile("test.txt", {}, function (file) {
        file.file(function (f) {
           var reader = new FileReader();
           reader.onloadend = function () {
              document.getElementById("txt").innerHTML =
```

```
     this.result;
         };
         reader.readAsText(f);
       });
 });
});
```

VIII. Les capteurs

L'un des intérêts d'un téléphone « intelligent », par rapport à un ordinateur portable (et même une tablette), est de posséder un ensemble de capteurs lui permettant de détecter son environnement immédiat. Parmi les capteurs les plus courants, nous allons trouver :

- La géolocalisation du téléphone
- Le capteur géomagnétique (boussole)
- Le capteur de luminosité
- Le capteur de son (microphone)
- Le capteur de mouvement (gyroscope, accéléromètre)

Tous les téléphones ne possèdent pas bien entendu tous les capteurs possibles ! Nous allons voir dans la suite de ce chapitre les capteurs les plus courants, et comment les utiliser dans les différents outils de développement vus au 0.

Sur chacun des systèmes mobiles et pour chaque capteur, une autorisation **explicite** de l'utilisateur est nécessaire pour qu'une application puisse l'utiliser.

1. Géolocalisation

La géolocalisation consiste à connaître sa position sur le globe terrestre. Les utilisations sont nombreuses : une assistance à la navigation par exemple, ou des services différents suivant la localisation.

Les coordonnées géographiques utilisées par les systèmes de géolocalisation sont des coordonnées **sphériques** (la terre étant, grosso modo, une sphère) au nombre de trois (dans un système à 3 dimensions il faut trois coordonnées).

La première des coordonnées est la **longitude**. Elle est définie par un angle par rapport à un méridien de référence (le méridien passant par Greenwich[1]), un **méridien** étant une ligne imaginaire parcourant la surface du globe pour relier le pôle nord au pôle sud. On exprime habituellement cet angle en degrés entre 0 et 180. On précise si on prend un méridien à l'est ou à l'ouest du méridien de référence.

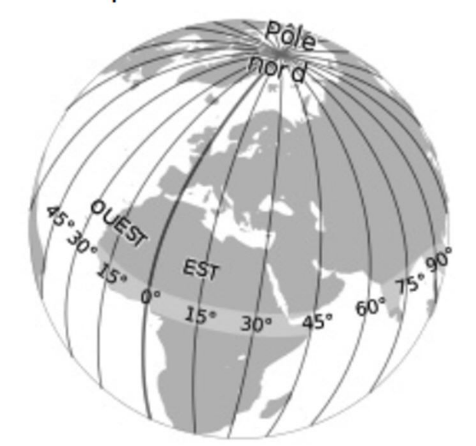


Figure 1 : mesure de la longitude

Les premières mesures de la longitude utilisaient la position des étoiles pour la nuit, et le décalage horaire entre l'heure solaire et l'heure locale pour le jour.

La deuxième coordonnée est la **latitude**. Ici aussi c'est une mesure d'angle par rapport au plan de l'équateur (ligne à mi-chemin entre les pôles). Ces lignes sont perpendiculaires aux méridiens et suivent également la surface du globe. La mesure s'exprime entre 0 (à l'équateur) et 90° (aux pôles), il faut donc préciser si l'on est dans l'hémisphère nord ou sud. Les premières mesures utilisaient un instrument (astrolabe, sextant ou gnomon) pour estimer l'angle avec une référence : à midi le soleil est à la verticale de l'équateur, on peut utiliser aussi une étoile connue la nuit.

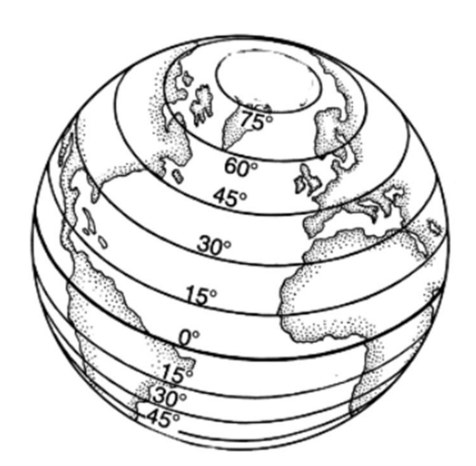


Figure 2 : mesure de la latitude

La troisième mesure est **l'altitude**, définie par la distance du point avec le niveau de référence (appelé niveau de la mer). Cette mesure n'est pas un angle et s'exprime en général en mètres.

Les téléphones utilisent plusieurs technologies pour estimer la géolocalisation.

La plus connue est également la plus précise : c'est la localisation par satellite. Le système le plus courant est le GPS (Global Positionning System)

---

[1] Ville d'Angleterre, proche de Londres, choisie comme méridien 0 car possédant un observatoire astronomique.



Figure 3: représentation des satellites GPS

américain, mais il existe également le système GLONASS (russe), GALILEO (européen) et BEIDOU (chinois). Tous utilisent le même principe : un ensemble de satellites à haute altitude (20 000 km environ) émettant un système radio que le téléphone récupère et utilise pour estimer la position. La précision dans les meilleurs conditions est de l'ordre du mètre. Un téléphone équipé d'une puce GPS sera donc capable, sous reserver de capter le signal des satellites (difficile à l'intérieur d'un batiment, d'un tunnel…), de calculer la position de manière très précise.

Un téléphone peut aussi utiliser les antennes GSM : en effet les antennes captées par le téléphone sont à des emplacements précis ; si le téléphone en capte trois, il est capable par triangularisation de connaitre sa position. La précision dépend de la densité des antennes relais : en ville la précision est de l'ordre de la centaine de mètres, alors qu'en campagne elle est de l'ordre de la dizaine de kilomètres.

Si le téléphone est équipé d'une puce Wifi et qu'il capte une borne wifi publique, la position de celle-ci étant connue le téléphone peut connaître sa position avec une précision de l'ordre de la portée du Wifi : 300 mètres.

Sur la plupart des systèmes, on pourra choisir entre une localisation précise (avec le GPS) ou imprécise (GSM, Wifi, par l'adresse IP…).

Système	Précision	Technologie
GPS	1m – 20m	Constellation de satellites
GSM	100m – 20 km	Position des antennes relais
Wifi	300m	Position de la borne Wifi

Tableau 11 : technologies de géolocalisation

a) Android natif

Le système Android fournit une API de géolocalisation qui utilise soit le GPS (si disponible), soit le GSM (si possible) soit le Wifi. Cette API est basée sur les classes suivantes :

- `LocationManager` : service système gérant les fournisseurs de services de localisation
- `LocationProvider` : fournisseur de géolocalisation
- `LocationListener` : interface spécifiée pour « écouter » la localisation
- `Location` : représente la position (les coordonnées).

A l'aide d'Android Studio, développons une petite application pour tester cette API.

Il faut tout d'abord donner à l'application l'autorisation d'utiliser le GPS. Pour cela, on édite le fichier `AndroidManifest.xml` et on ajoute la ligne :

```
<uses-permission
    android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

L'application ne contient qu'une seule activité (un écran) qui englobera trois `TextView` (vues textes) appelées `latitude`, `longitude`, `altitude`.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android
"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"

android:paddingBottom="@dimen/activity_vertical_margin"

android:paddingLeft="@dimen/activity_horizontal_margin"

android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"

tools:context="com.example.alexandre.androidloc.MainActiv
ity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"

android:textAppearance="?android:attr/textAppearanceLarge
"
        android:text="..."
        android:id="@+id/altitude"
        android:layout_alignParentTop="true"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"

android:textAppearance="?android:attr/textAppearanceLarge
"
        android:text="..."
        android:id="@+id/longitude"
        android:layout_below="@+id/altitude"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_marginTop="46dp" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"

android:textAppearance="?android:attr/textAppearanceLarge
"
        android:text="..."
        android:id="@+id/latitude"
        android:layout_below="@+id/longitude"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_marginTop="67dp" />
</RelativeLayout>
```

Capture 44 : code XML de l'activité de géolocalisation

La classe de l'activité doit donc avoir comme attributs trois instances de la classe `TextView` (reliées aux contrôles visuels) et doit implémenter l'interface `LocationListener` :

```
public class MainActivity extends AppCompatActivity
    implements LocationListener {
        private TextView latitude;
```



```
        private TextView longitude;
        private TextView altitude;
```

Dans la fonction redéfinie `onLocationChanged` (appelée par le fournisseur de localisation dès que nécessaire), il suffit de lire la localisation et de la transmettre aux contrôles visuels :

```
@Override
public void onLocationChanged(Location location) {
    latitude.setText(String.valueOf(location.getLatitude()))
    ;
    altitude.setText(String.valueOf(location.getAltitude()))
    ;
    longitude.setText(String.valueOf(location.getLongitude()
    ));
}
```

Il faut initialiser les attributs dans la fonction `onCreate` :

```
latitude = (TextView)findViewById(R.id.latitude);
altitude = (TextView)findViewById(R.id.altitude);
longitude = (TextView)findViewById(R.id.longitude);
```

Pour simplifier, créons une fonction privée pour initialiser la lecture régulière de la position (la valeur 100 indique que nous voulons une mise à jour toutes les 100 millisecondes, la valeur 1 que nous voulons une mise à jour si la position change de 1 m) :

```
private void initLocation() throws  SecurityException
{
LocationManager manager =
    (LocationManager)getSystemService(LOCATION_SERVICE);
    manager.requestLocationUpdates(LocationManager.GPS_PROVI
    DER, 100, 1, this);
}
```

Il reste à faire le plus complexe : initier la connexion au fournisseur de localisation. Il est obligatoire que l'utilisateur accepte explicitement l'utilisation pour chaque application de la localisation. On va par conséquent vérifier, dans le code de la fonction `onCreate`, si l'application possède déjà l'autorisation. Si c'est le cas, on peut appeler la fonction privée ci-dessous :

```
int ret = ContextCompat.checkSelfPermission(this,
    Manifest.permission.ACCESS_FINE_LOCATION);
if( ret == PackageManager.PERMISSION_GRANTED)
{
```

```
    initLocation();
}
```

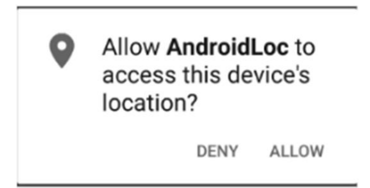
Mais si cependant l'autorisation n'a pas encore été accordée, il est nécessaire de la demander à l'utilisateur par le code suivant (100 est un entier quelconque qui sert à identifier la requête) :

```java
else
{
    ActivityCompat.requestPermissions(this,
    new
    String[]{Manifest.permission.ACCESS_FINE_LOCATION},100);
}
```

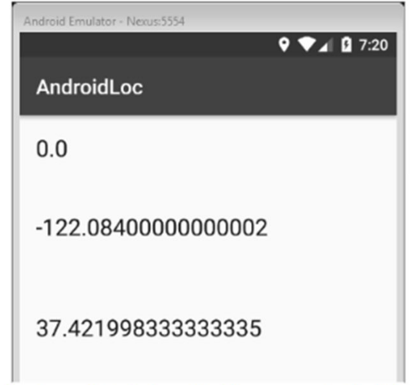
De manière asynchrone, une boite de dialogue va surgir pour demander à l'utilisateur l'autorisation. S'il l'accepte, l'opération `onRequestPermissionsResult` de la classe d'activité est appelée, il faut donc la redéfinir :

```java
@Override
public void onRequestPermissionsResult(int requestCode,
    String[] permissions, int[] results)
{
    initLocation();
}
```

Lors de la première exécution de l'application, l'autorisation d'utiliser la localisation sera demandée à l'utilisateur ; s'il l'accepte, la question ne lui sera plus posée, sauf s'il révoque cette autorisation dans les paramètres du téléphone.



Capture 45 : demande d'autorisation d'utiliser la localisation sous Android



Capture 46 : la localisation sous Android

Bien entendu, l'affichage n'est pas très visuel, ni lisible, les coordonnées étant plutôt exprimées en général sous la forme degré/minutes/secondes (par exemple : 47°12'45") mais la conversion est facile à faire et n'est pas spécifique à une plateforme ou un langage.

b) iOS natif

Ouvrez l'application XCode et créez un nouveau projet pour iOS, de type « *single view application* », en utilisant le langage *Swift* (la même chose est possible en Objective-C mais *Swift* est à la fois plus moderne et plus lisible). Il faut **signer** l'application avec votre *Apple ID*.



Commençons par éditer le *storyboard*.

Dans le fichier du contrôleur du *storyboard* (par défaut `ViewController.swift`) la première ligne à ajouter sert à importer le module de géolocalisation :

```
import CoreLocation
```

Un attribut pour le gestionnaire de localisation est nécessaire :

```
private var locationManager = CLLocationManager()
```

La vue doit implémenter l'interface appropriée :

```
class ViewController: UIViewController,
    CLLocationManagerDelegate {
```

et initialiser la vue :

```
override func viewDidLoad()
{
    super.viewDidLoad()
    self.locationManager.delegate = self
```

Swift

```
    self.locationManager.requestAlwaysAuthorization()
    self.locationManager.desiredAccuracy =
    kCLLocationAccuracyBest
    self.locationManager.startUpdatingLocation()
```

Pour demander l’autorisation, depuis iOS 8, il est nécessaire d’ajouter dans le fichier `info.plist` les valeurs suivantes :

```
<key>NSLocationAlwaysUsageDescription</key>
<string>L'application a besoin de la localisation</string>
<key>NSLocationWhenInUseUsageDescription</key>
<string> L'application a besoin de la localisation </string>
```

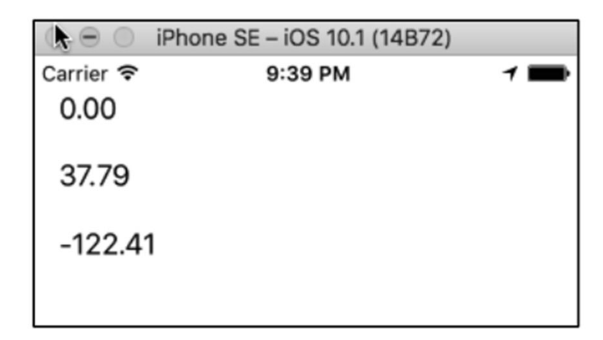
L’étape suivante consiste à relier les contrôles de la vue au contrôleur, en créant des *outlets*[1]. Il suffit de glisser-déposer (tout en maintenant la touche CTRL) un contrôle depuis la vue vers le code (en mode assistant) pour créer les trois *outlets* :

```
@IBOutlet weak var altitude : UILabel!
@IBOutlet weak var latitude : UILabel!
@IBOutlet weak var longitude : UILabel!
```

Il ne reste plus qu’à implémenter le délégué pour répondre à la modification de la localisation et afficher dans la vue le résultat :

```
func locationManager(_ manager :CLLocationManager,
    didUpdateLocations locations :[CLLocation]) {
    altitude.text =
      String(format :"%.2f",(locations.last?.altitude)!)
    latitude.text =
    String(format :"%.2f",(locations.last?.coordinate.latitu
    de)!)
    longitude.text =
      String(format :"%.2f",
    (locations.last?.coordinate.longitude)!)
}
```

[1] *Outlet* : pour iOS, connexion entre un contrôle visuel et une variable dans le contrôleur.



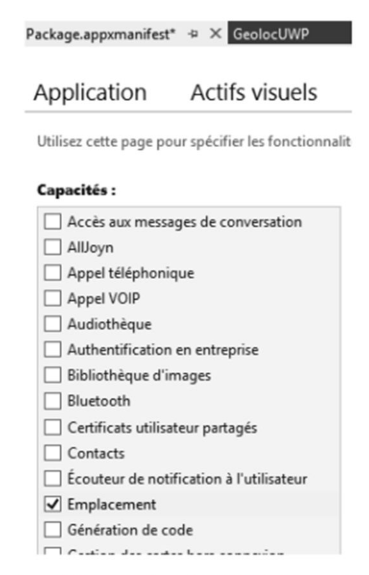
Capture 47 : géolocalisation avec XCode dans émulateur iOS

c) Windows 10 natif

Windows UWP propose des API simples pour gérer la localisation, basées sur les classes suivantes :

- `Geolocator` : fournisseur
- `Geoposition` : coordonnées

L'application doit indiquer dans son manifeste l'utilisation de la géolocalisation.



Capture 48 : autorisation d'une application UWP à accéder à l'emplacement

Un simple écran, composé de 3 zones de texte, suffit pour tester la géolocalisation :




```
<Page
    x:Class="GeolocUWP.MainPage"

       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/p
       resentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:GeolocUWP"

       xmlns:d="http://schemas.microsoft.com/expression/blen
       d/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-
       compatibility/2006"
    mc:Ignorable="d"
    FontSize="24" Height="492" Width="266"    >

    <Grid Background="{ThemeResource
       ApplicationPageBackgroundThemeBrush}">
        <Grid.RowDefinitions>
            <RowDefinition Height="*"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <TextBlock Grid.Row="0" Text="Altitude"
       x:Name="altitude" Margin="10" />
        <TextBlock Grid.Row="1" Text="Latitude"
       x:Name="latitude"  Margin="10"/>
        <TextBlock Grid.Row="2" Text="Longitude"
       x:Name="longitude"  Margin="10"/>
    </Grid>
</Page>
```

Dans une opération, nous demandons l’autorisation d’utiliser la localisation. Si nous l’obtenons, il faut créer une instance du `Geolocator` en indiquant l’intervalle de report (ici 1000 ms), la précision désirée (ici haute, correspondant au GPS) et associer l’évènement `PositionChanged` à une opération de la classe :

```
async void init()
{
    var status = await Geolocator.RequestAccessAsync();
    if(status == GeolocationAccessStatus.Allowed)
    {
      Geolocator geo = new Geolocator()
      {
        ReportInterval = 1000,
        MovementThreshold = 1,
        DesiredAccuracy = PositionAccuracy.High
      };
      geo.PositionChanged += Geo_PositionChanged;
    }
}
```

Passons à présent à la fonction de réponse à l’évènement qui sera chargée, de manière asynchrone, d’afficher le résultat. Il faut faire attention

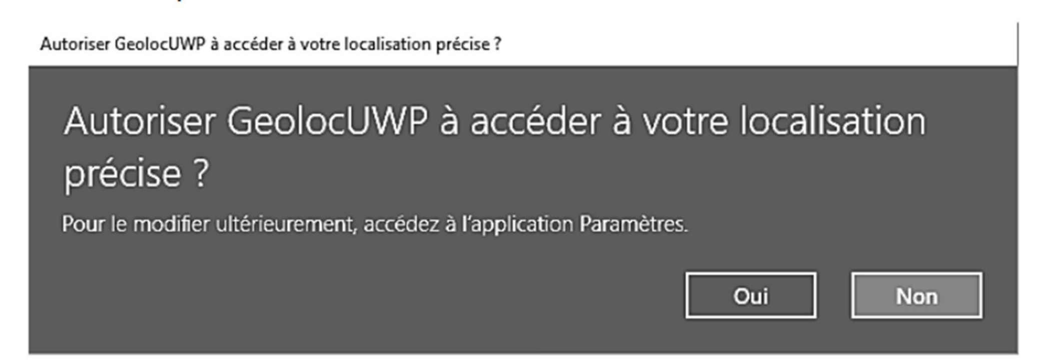
ici : la fonction est appelée depuis un autre *thread*[1], or Windows impose que l'affichage se fasse uniquement dans le *thread* principal, il faut donc placer le code dans une fonction `RunAsync` qui se chargera de faire exécuter celui-ci dans le bon *thread*.

```
private async void Geo_PositionChanged(Geolocator sender,
     PositionChangedEventArgs args)
{
     Geoposition pos = args.Position;
     await Dispatcher.RunAsync(
     Windows.UI.Core.CoreDispatcherPriority.Normal, () =>
        {
           altitude.Text =
     pos.Coordinate.Point.Position.Altitude.ToString();
           latitude.Text =
     pos.Coordinate.Point.Position.Latitude.ToString();
           longitude.Text =
     pos.Coordinate.Point.Position.Longitude.ToString();
     });
}
```

Il suffit d'appeler la fonction `init()` dans le constructeur de la page.

Bien sûr, pour que tout fonctionne il faut que l'accès à la localisation soit activé dans le système.

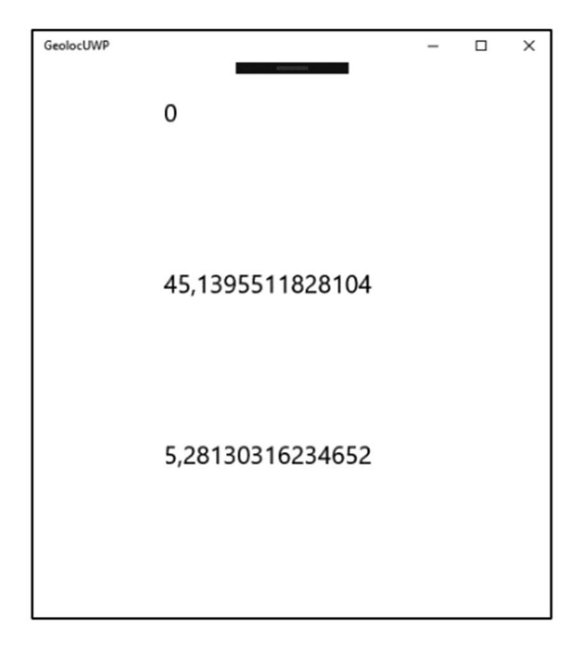
Lors de la première exécution l'autorisation est demandée à l'utilisateur :

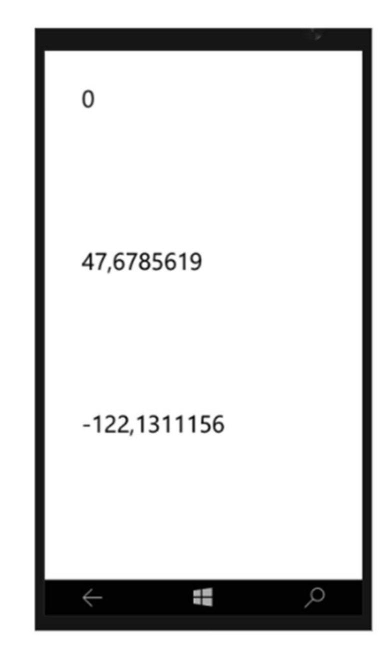


L'autorisation ne sera ensuite plus demandée. L'application peut être testée directement sous Windows (ordinateur de bureau) ou dans un

[1] Un *thread* est une unité d'exécution, une partie d'une application qui s'exécute en parallèle des autres, ce qui permet de ne pas bloquer l'application lors d'un traitement long (par exemple, calculer la géolocalisation).

émulateur de téléphone Windows, ou directement dans le téléphone relié au PC.





Capture 49 : géolocalisation avec Windows 10, tablette et mobile

d) Multiplateforme - Qt

Le *framework* Qt contient une API pour la géolocalisation. Il faut ajouter dans le fichier du projet la ligne suivante :

```
QT += positioning
```

Et inclure dans le fichier source l'entête *ad hoc* :

```
#include <QGeoCoordinate>
```

L'API est fonctionnelle pour Android, iOS, macOS, Linux, Windows RT, soit pour la plupart des systèmes sur téléphone.

La classe principale de cette API est `QGeoPositionInfo` qui contient les coordonnées ainsi que l'heure précise de la mesure. Dans certains cas, la classe contient aussi les informations liées à la vitesse et à la précision de la mesure.

Pour obtenir la position, il faut également utiliser la classe `QGeoPositionInfoSource`.

Nous allons écrire une petite application simple pour afficher les coordonnées courantes sur l'écran.

L'écran sera très simple, composé de 3 contrôles `Label` nommés `altitude`, `latitude` et `longitude`.

Dans le fichier d'entête (`.h`) de la fenêtre, il faut ajouter un *slot*[1] pour recevoir la notification de modification de position ainsi qu'un attribut de type `QGeoPositionInfoSource` :

```
private slots:
    void nouvellePosition(const QGeoPositionInfo& info);
private:
    QGeoPositionInfoSource* source;
```

Dans le fichier source (`.cpp`) de la fenêtre, il faut dans le constructeur de celle-ci récupérer l'objet gérant la localisation et le relier au *slot* défini ci-dessus :

```
source = QGeoPositionInfoSource::createDefaultSource(this);
if(source)
{
     connect(source,
     SIGNAL(positionUpdated(QGeoPositionInfo)),this,
     SLOT(nouvellePosition(QGeoPositionInfo)));
     source->startUpdates();
}
else
{
     QMessageBox::warning(this,"Erreur",
     "Pas de capteur de position");
}
```

La fonction suivante est le *slot* Qt qui sera automatiquement appelé quand la position change :

```
void MainWindow::nouvellePosition(const QGeoPositionInfo
     &info)
{
    if(info.coordinate().isValid())
    {
```

[1] *Slot* Qt : système de Qt pour faire de la programmation évènementielle. Un *slot* est une fonction membre de la classe de fenêtre qui est un récepteur pour un message envoyé par une autre partie de l'application.

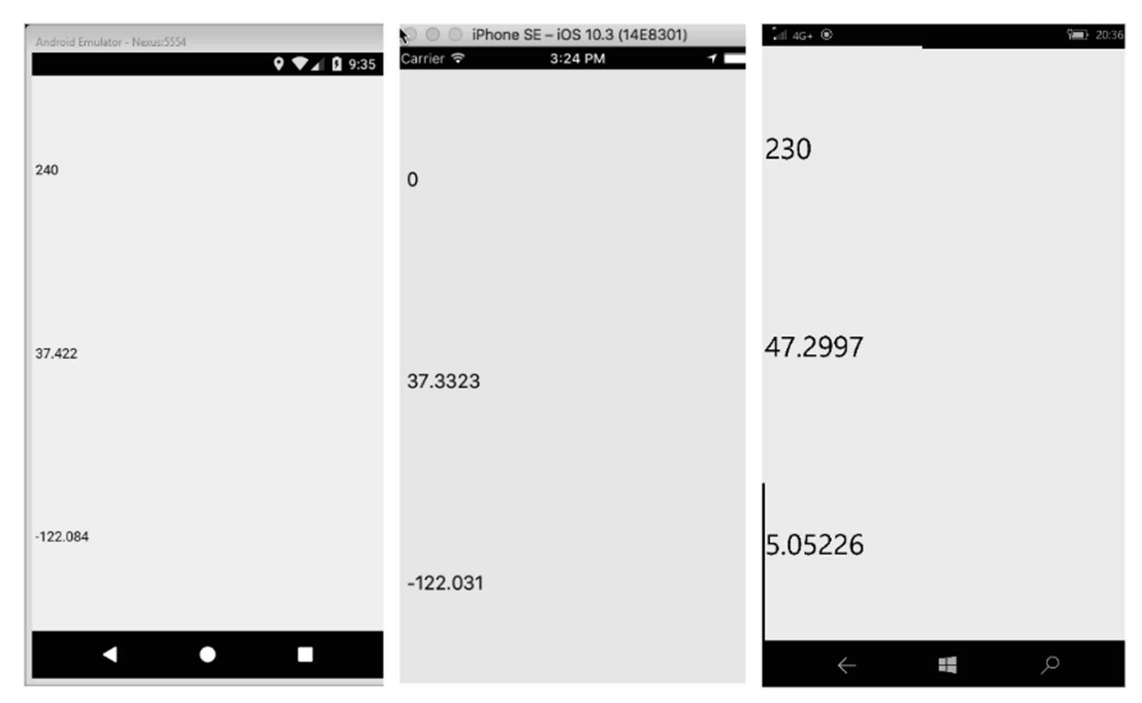
```
        double longitude = info.coordinate().longitude();
        double latitude = info.coordinate().latitude();

        ui->altitude->setText(QString::number(altitude));
        ui->latitude->setText(QString::number(latitude));
        ui->longitude->setText(QString::number(longitude));
    }
}
```

Le code créé est le même sur les hôtes Windows, Mac OS et Linux, que la cible soit un iPhone, un téléphone Android, un téléphone Windows.

Pour iOS il faut ajouter dans le fichier `Info.plist` (voir IV.4.e) les lignes suivantes pour avoir l'autorisation d'utiliser la localisation :

```
<key>UIRequiredDeviceCapabilities</key>
<array>
     <string>location-services</string>
     <string>gps</string>
</array>
<key>NSLocationAlwaysUsageDescription</key>
<string>Test pour la localisation avec QT</string>
<key>NSLocationWhenInUseUsageDescription</key>
<string>Test pour la localisation Qt</string>
```



Capture 50 : géolocalisation avec Qt pour Android, iOS, Windows 10 mobile

e) Multiplateforme - Xamarin

D'origine, *Xamarin* ne fournit pas d'API commune pour la géolocalisation, mais il est possible d'installer un *plugin*[1] à Visual Studio pour obtenir une API commune (16). Nous pouvons donc utiliser *Xamarin.Forms* (pour avoir une interface utilisateur commune) et cette API dans un seul projet pour iOS, Android, Windows.

Créez une solution « *cross-platform* » avec *Xamarin.Forms* et un projet partagé (ne pas utiliser la bibliothèque portable PCL, le *plugin* n'est pas compatible avec elle dans sa version actuelle[2]). Installez le *plugin* pour la géolocalisation dans le **projet commun** en utilisant le gestionnaire de *packages NuGet* (menu « *outils → gestionnaire de packages → gérer les packages nuget pour la solution* »). Bien lire le fichier `readme.txt` à la fin de l'installation : il précise comment utiliser correctement la localisation sur chaque plateforme. La documentation de ce *plugin* est disponible (17).

Commençons par définir la page principale (fichier `MainPage.xaml` du projet partagé) en déposant simplement trois *labels* :

```
<ContentPage><ContentPage.Content>
  <StackLayout>
   <Label Text="Altitude"
     x:Name="altitude" />
   <Label Text="Latitude"
     x:Name="latitude" />
   <Label Text="Longitude"
     x:Name="longitude"/>
  </StackLayout>
 </ContentPage.Content></ContentPage>
```

XAML

Altitude

Latitude

Longitude

Capture 51 : page principale, Géolocalisation xamarin

Dans le code de cette page (fichier `MainPage.xaml.cs`), Commençons par importer les espaces de nom du *plugin* :

```
using Plugin.Geolocator;
```

Il faut ensuite créer une fonction qui démarre une écoute sur la position (ici avec un intervalle de 2s et une précision de 5 m) :

---

[1] Voir les instructions sur https://www.nuget.org/packages/Xam.Plugin.Geolocator.

[2] Version 4.1.2 à l'heure où ces lignes sont imprimées.

Il faut ensuite créer une fonction qui démarre une écoute sur la position (ici avec un intervalle de 2s et une précision de 5 m) :

```
private async void init() {
     await CrossGeolocator.Current.StartListeningAsync(
     TimeSpan.FromSeconds(2), 5);
     CrossGeolocator.Current.PositionChanged += MajPosition;
}
```

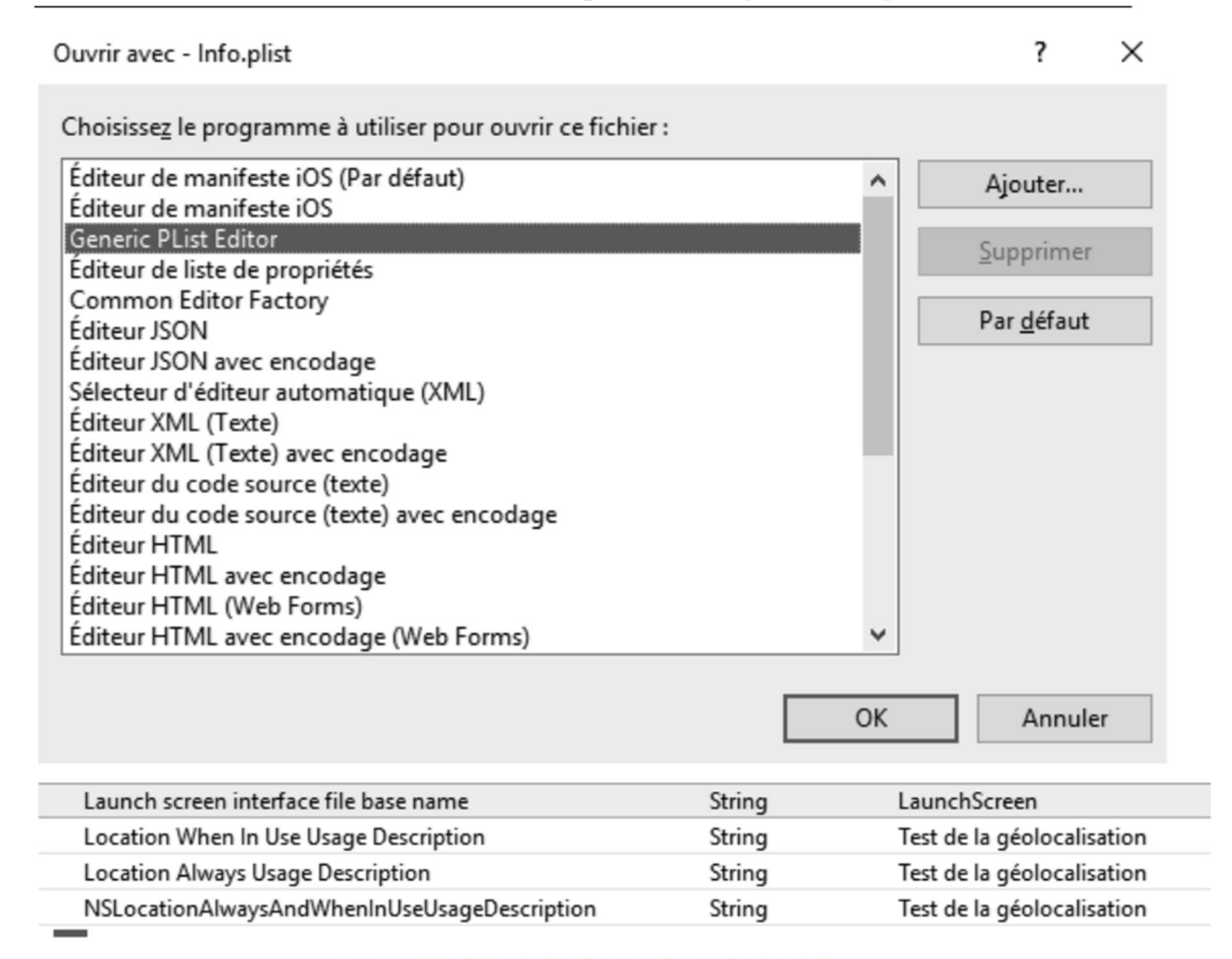
Cette fonction sera appelée dans le constructeur de la page.

La fonction appelée à chaque modification de la position est la suivante. Elle modifie l'affichage dans un *thread* différent du *thread* principal et doit donc utiliser la fonction `BeginInvokeOnMainThread` :

```
private void MajPosition(object sender,
     Plugin.Geolocator.Abstractions.PositionEventArgs e) {
     Device.BeginInvokeOnMainThread(() => {
        altitude.Text = e.Position.Altitude.ToString();
        latitude.Text = e.Position.Latitude.ToString();
        longitude.Text = e.Position.Longitude.ToString();
     });
}
```

Pour Windows UWP, il est nécessaire d'**autoriser** l'emplacement (la lecture de la position) dans le manifeste (fichier `Package.appxmanifest`) et installer le *package NuGet* dans le projet UWP.

Pour iOS il faut ajouter dans le fichier `Info.plist` des clés expliquant pourquoi l'application utilise la localisation, ce qui peut se faire dans Visual Studio en utilisant « *Generic Plist Editor* » (bouton droit sur `Info.plist` dans la fenêtre de projet, puis « ouvrir avec ») :



Capture 52 : fichier Info.plist pour la localisation iOS

Pour Android, le niveau 25 d'API est obligatoire (pour le *plugin*) et il est nécessaire d'ajouter dans la classe de l'activité principale (fichier `MainActivity.cs` par défaut) la fonction suivante :

```
public override void OnRequestPermissionsResult(int
     requestCode, string[] permissions, Permission[]
     grantResults)
{
     PermissionsImplementation.Current.OnRequestPermissionsRe
     sult(requestCode, permissions, grantResults);
}
```







Capture 53 : géolocation avec Xamarin sous Windows 10 mobile, iOS, Android

f) Multiplateforme - Cordova / HTML5

Javascript possède des API utilisables pour accéder à la localisation, notamment :

- L'objet `geolocation`, propriété de l'objet `navigator` défini globalement, qui permet d'accéder aux coordonnées (objet `coords`)
- La fonction `getCurrentPosition` qui permet de lire (si l'utilisateur l'autorise) la position actuelle

Si l'on souhaite lire régulièrement la position, il suffit d'utiliser la fonction Javascript `setInterval`.

Le projet Cordova peut être créé avec Visual Studio 2017, mais ce n'est pas obligatoire.

Le projet doit contenir au moins les fichiers suivants :

- Un fichier HTML définissant le contenu de la page (écran)
- Un fichier CSS définissant les styles utilisés
- Un fichier JS (javascript) contenant le code nécessaire

Voici un extrait du code HTML définissant la page (la CSS utilisée est celle par défaut, mais vous pouvez adapter à votre goût) :

```html
<body>
     <div class="app">
        <h1>Géolocalisation</h1>
           <p>Altitude : <span id="altitude"></span></p>
           <p>Latitude : <span id="latitude"></span></p>
           <p>Longitude : <span id="longitude"></span></p>
     </div>
     <script type="text/javascript"
     src="cordova.js"></script>
     <script type="text/javascript"
     src="scripts/platformOverrides.js"></script>
     <script type="text/javascript"
     src="scripts/index.js"></script>
</body>
```

La fonction Javascript pour afficher la géolocalisation :

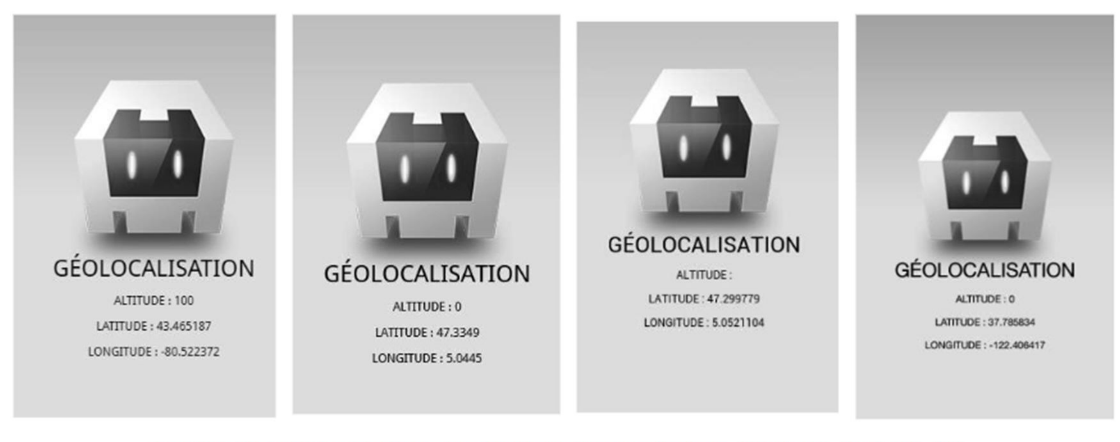
```javascript
function onLocationOk(pos) {
     var altitude = document.getElementById("altitude");
     altitude.innerHTML = pos.coords.altitude;
     var latitude = document.getElementById("latitude");
     latitude.innerHTML = pos.coords.latitude;
     var longitude = document.getElementById("longitude");
     longitude.innerHTML = pos.coords.longitude;
}
```

Il ne reste plus qu'à ajouter, dans la fonction `onDeviceReady` (appelée dès que l'application est chargée), l'appel régulier à cette fonction grâce à `setInterval` (ici l'appel se fera toutes les secondes) :

```javascript
setInterval(function () {
     navigator.geolocation.getCurrentPosition(onLocationOk);
     }, 1000);
```

Il faut évidemment inclure la demande d'autorisation pour l'utilisation de la géolocalisation. Si vous incluez le plugin `cordova.plugin.geolocation`, cela se fait directement. Pour insérer ce plugin il suffit d'ouvrir le fichier `config.xml` et de choisir l'onglet *plugins*, puis d'installer le *plugin* `geolocation`.

Le résultat ressemble aux captures suivantes (remarquez que l'apparence change peu d'un système à l'autre) :



Capture 54 : géolocalisation avec Cordova dans navigateur, Windows, Android, iOS

2. Appareil photo

La quasi-totalité des téléphones possèdent un appareil photo, avec une qualité qui est équivalente (voire dans certains cas supérieure) aux appareils photos numériques compacts. Un tel capteur est très pratique pour prendre des photos, des vidéos (couplé avec un microphone pour le son), déchiffrer des codes-barres, etc. Certains appareils possèdent même plusieurs capteurs (par exemple, un en face avant et un en face arrière). L'utilisation d'un tel capteur est classique et essentielle pour la programmation d'applications sur mobiles.

Nous allons voir dans cette partie comment utiliser l'appareil photo dans une application simple, sur les différents outils abordés.

a) Android natif

Le moyen le plus simple de prendre une photo est d'utiliser l'application native Android. Il est bien entendu possible de gérer directement dans son application la caméra mais cela réclame beaucoup plus de code. De plus, les API de gestion de la caméra sont différentes suivant la version de l'API Android utilisé. Pour faire simple, nous utiliserons donc une capture photo via l'application dédiée. Outre la simplicité, cette solution présente l'avantage de ne pas avoir besoin d'ajouter une autorisation dans le manifeste : ce n'est pas notre application qui prend la photo.

Dans notre classe d'activité, nous allons déposer un simple bouton pour lancer la capture, et une `ImageView` pour afficher le résultat.

Dans l'opération `onCreate` de l'activité, commençons par relier le bouton avec un code de déclenchement :

```
Button b = (Button)findViewById(R.id.button);
b.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            takePhoto();
        }
});
```

Créons ensuite cette opération `takePhoto`. Dès qu'il faut passer d'une activité à l'autre, les API Android utilisent la classe `Intent`. Dans notre cas, nous allons utiliser `l' intent` du système dédié à la capture photo :

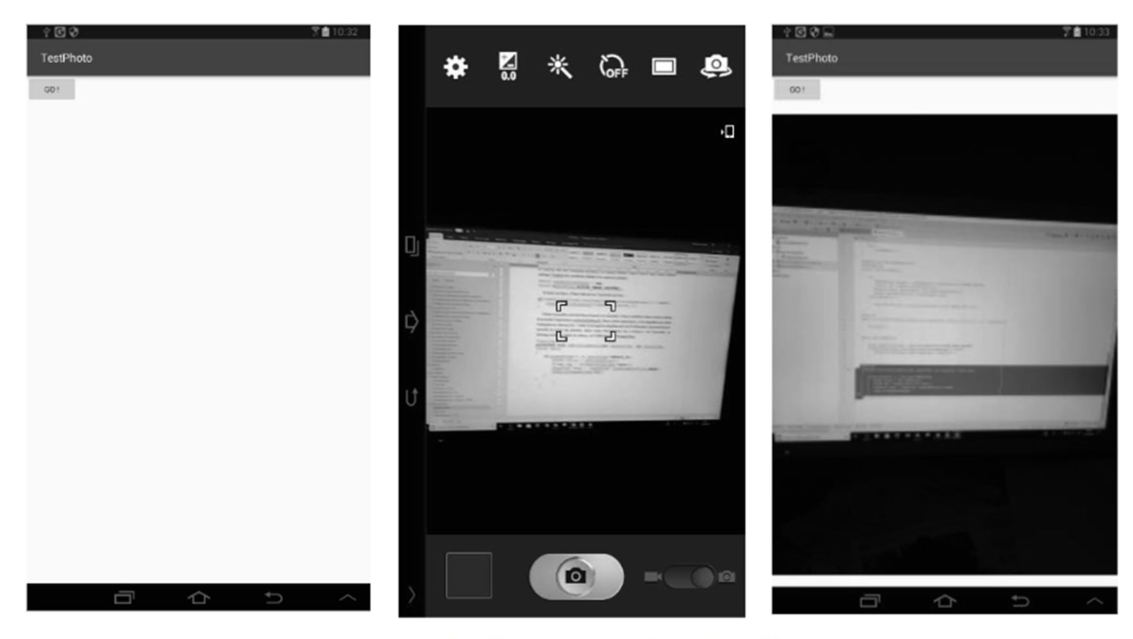
```
Intent takePictureIntent = new
     Intent(MediaStore.ACTION_IMAGE_CAPTURE);
```

Si tout va bien, il faut démarrer l'activité *ad hoc* :

```
if(takePictureIntent.resolveActivity(getPackageManager())!=nu
     ll){
        startActivityForResult(takePictureIntent,1);
}
```

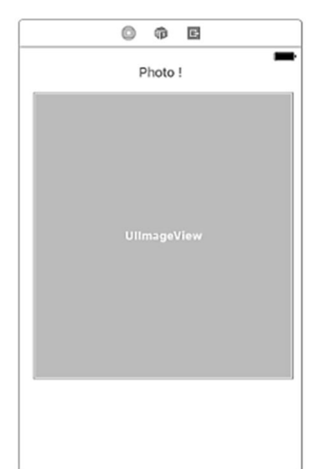
Cette nouvelle activité fournissant un résultat, il faut redéfinir dans notre classe d'activité l'opération `onActivityResult`. Dans cette opération, si la requête est celle indiquée au-dessus (ici : code 1) et que le résultat est « ok » (l'utilisateur pouvant avoir annulé la prise de photo), alors nous récupérons les « extras » de l'activité, le *bitmap* contenu dans le retour, et l'affectons à `l' ImageView`.

```
@Override
protected void onActivityResult(int requestCode, int
     resultCode, Intent data)
{
     if(requestCode==1 && resultCode==RESULT_OK){
        Bundle extras = data.getExtras();
       Bitmap bmp = (Bitmap)extras.get("data");
       ImageView image = (ImageView)
     findViewById(R.id.image);
       image.setImageBitmap(bmp);
     }
}
```



Capture 55 : étapes capture photo Android

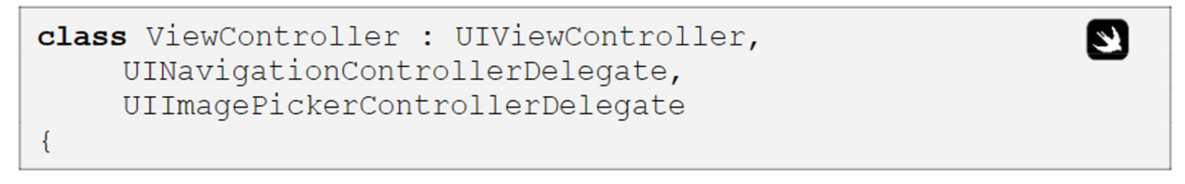
b) iOS natif



Nous allons réaliser une simple application pour prendre une photo. Commençons par créer une application (*Single View Application*) en Swift. Editez le *storyboard* pour y placer un contrôle `UIImageView` et un contrôle `UIButton`.

Créez un *outlet* (ctrl+glisser) avec le contrôle `ImageView` appelé `imageView`. Créez ensuite une action (ctrl+glisser) avec le bouton, appelée `takePhoto`.

Editez le code du contrôleur de vue (`ViewController.swift`) ; votre classe de contrôleur doit hériter de `UINavigationControllerDelegate` et de `UIImagePickerControllerDelegate`.



```
class ViewController : UIViewController,
     UINavigationControllerDelegate,
     UIImagePickerControllerDelegate
{
```

Déclarez ensuite un attribut de type `UIImagePickerController` :

```
var imagePicker = UIImagePickerController()
```



Dans l'action liée au bouton, entrez le code suivant :

```
@IBAction func takePhoto(sender: UIButton) {
    imagePicker =  UIImagePickerController()
    imagePicker.delegate = self
    imagePicker.sourceType = .camera
    present (imagePicker, animated: true, completion: nil)
  }
```

Swift

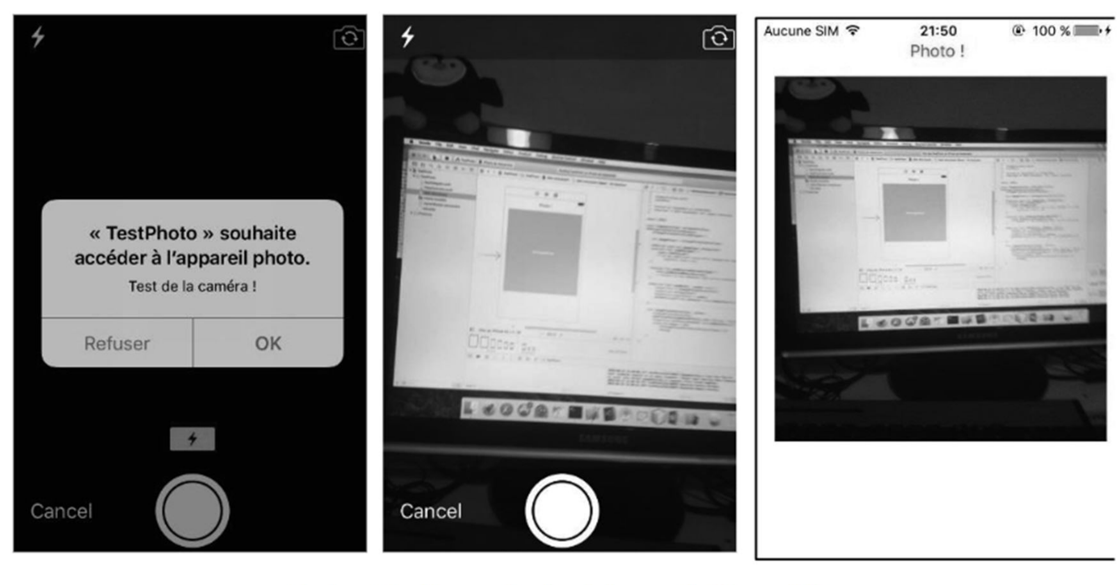
Il faut ensuite redéfinir une opération :

```
func imagePickerController(picker: UIImagePickerController,
     didFinishPickingMediaWithInfo info: [String : Any]) {
    imagePicker.dismiss (animated :true, completion: nil)
    imageView.image =
     info[UIImagePickerControllerOriginalImage] as? UIImage
  }
```

Il faut enfin demander l'autorisation d'utiliser la caméra. Editez le manifeste (`info.plist`) et rajoutez (bouton +) à la fin un texte (quelconque) pour le paramètre « » :

▶ Supported interface orientations	Array	(3 items)
▶ Supported interface orientations (i...	Array	(4 items)
Privacy - Camera Usage Description	String	Test de la caméra !

Et il reste plus qu'à tester l'application (sur un vrai iPhone, le simulateur iOS ne possédant pas de caméra) ; lors de la première utilisation, l'autorisation sera demandée à l'utilisateur :



Capture 56 : étapes prise de photo avec Swift pour iOS

c) Windows natif

Il est possible d'accéder à la caméra directement ou en utilisant l'application fournie par le système. Cette deuxième solution est plus simple et ne nécessite pas d'autorisations particulières dans l'application.

La première étape est d'instancier un objet de type `CameraCaptureUI` :

```
CameraCaptureUI captureUI = new CameraCaptureUI();
```

La deuxième étape est de définir le format désiré pour l'image (jpeg ou png) :

```
captureUI.PhotoSettings.Format =
    CameraCaptureUIPhotoFormat.Jpeg;
```

Il suffit de lancer la capture, celle-ci va récupérer l'image capturée, la sauvegarder dans le dossier local de l'application, et fournir une instance de `StorageFile` :

```
StorageFile file = await
    captureUI.CaptureFileAsync(CameraCaptureUIMode.Photo);
```

Ce fichier peut ensuite être déplacé, copié, ou simplement affiché dans un contrôle `Image` à l'écran :

```
var flux = await file.OpenReadAsync();
BitmapImage bmp = new BitmapImage();
await bmp.SetSourceAsync(flux);
image.Source = bmp;
```

d) Multiplateforme – Qt

Pour l'utilisation du capteur photo avec le *framework* Qt, le paquet « multimédia » est nécessaire, notamment la classe `QCamera`.

Il faut commencer donc par ajouter dans le projet l'utilisation des paquets multimédia :

```
QT += multimedia multimediawidgets
```

Et dans chaque fichier C++ utilisant la classe `QCamera`, inclure certains entêtes :

```
#include <QCamera>
#include <QCameraInfo>
#include <QCameraViewfinder>
```



Nous allons donc créer une application avec une seule fenêtre qui contiendra un interrupteur à bascule (ou case à cocher) pour visualiser ou non une caméra, une liste déroulante dans laquelle seront stockées les caméras existantes, une zone pour visualiser ce que « voit » la caméra, un bouton pour prendre une photo et sauvegarder l'image.

Nous aurons besoin d'un certain nombre d'attributs dans la classe de la fenêtre :

```
QList<QCameraInfo> camerasInfo; QCamera* camera=nullptr;
QCameraViewfinder *finder;
```

Avant toute chose, il faut récupérer la liste des caméras existantes, dans le constructeur de la fenêtre, pour initialiser la liste déroulante, et la stocker dans l'attribut `camerasInfo` :

```
camerasInfo = QCameraInfo::availableCameras();
foreach(const QCameraInfo& cam, camerasInfo)  {
        ui->cameras->addItem(cam.description());
}
```

Lorsque l'utilisateur choisit une caméra dans la liste (signal `currentIndexChanged`), on utilise les informations données par la classe `QCameraInfo` (le nom des caméras, leur position, etc…) pour créer la caméra. Cet objet est relié à la zone d'affichage :

```
void MainWindow::on_cameras_currentIndexChanged(int index)
{
     camera = new QCamera(camerasInfo[index]);
     finder = new QCameraViewfinder;
     ui->centralWidget->layout()->addWidget(finder);
     camera->setViewfinder(finder);
}
```

Lorsque l'utilisateur bascule l'interrupteur, on active ou non la prévisualisation de la caméra (celle-ci doit évidemment avoir été choisie).

```
void MainWindow::on_checkBox_toggled(bool checked)
{
    if(checked && camera)
    {
        finder->show();
        camera->start();
    }
    else      {
        camera->stop();
        finder->hide();
    }
}
```

Lors du clic sur le bouton, une photo est capturée, en utilisant un objet de type `QCameraImageCapture` :

```
void MainWindow::on_pushButton_clicked() {
     QCameraImageCapture *capt = new
     QCameraImageCapture(camera);
     capt->
setCaptureDestination(QCameraImageCapture::CaptureToBuffer);
     connect(capt,SIGNAL(imageCaptured(int,QImage)),this,
      SLOT(imageCaptured(int,QImage)));
     camera->searchAndLock();
     capt->capture();
     camera->unlock();
}
```

Il faut définir un *slot* dans la classe pour récupérer la capture de l'image (la fonction est asynchrone) :

```cpp
private slots:
    void imageCaptured(int id, QImage img);
```

Une fois l'image capturée il ne reste plus qu'à la sauvegarder si l'utilisateur le souhaite :

```cpp
void MainWindow::imageCaptured(int id, QImage img) {
    camera->stop();
    QString file = QFileDialog::getSaveFileName(this,
       "Sauvegarde de l'image");
    img.save(file);
    QMessageBox::information(this,"Terminé","Image sauvée");
    camera->start();
}
```

Suivant le cas, il se peut que l'image capturée par la caméra soit mal orientée.

Sur la majorité des téléphones, la première exécution provoquera l'apparition d'une boite de dialogue demandant l'autorisation d'utiliser la caméra (autorisation qu'il faut accorder, bien sûr !).

L'exécution donne les résultats suivants (les photos sont volontairement floutées) :





Avec Android et QWidgets, l'aperçu de photo ne fonctionne pas avec la version actuelle. Il faut donc soit utiliser QML, soit attendre une version qui fonctionne…

Capture 57 : Photo avec Qt sous iOS, Windows bureau

e) Multiplateforme - Xamarin

Pour conserver un aspect totalement multiplateforme, il est intéressant de recourir à un *plugin* pour *Xamarin* permettant d'utiliser la caméra du téléphone de manière identique pour iOS, Android, Windows 10. Disponible sur (16), le *plugin* `Xam.Plugins.Media` est tout indiqué pour ce travail. Voir page 176 pour l'installation d'un *plugin* dans la solution.

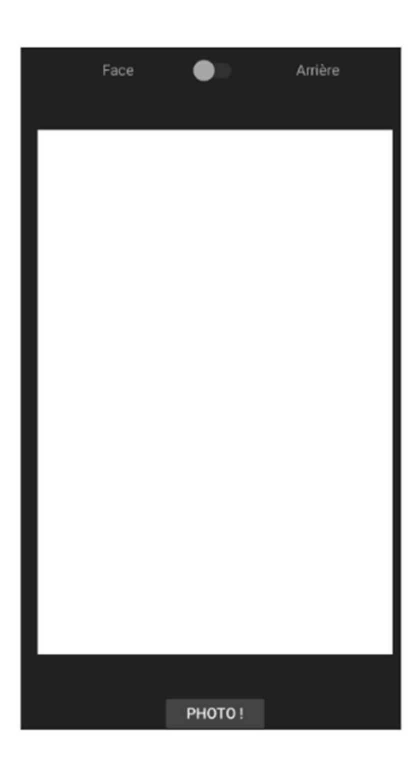
Nous allons créer une solution « *cross-platform* » en utilisant *Xamarin.Forms* et un projet partagé.

Commençons par installer ce *plugin* dans notre solution Visual Studio par *NuGet*. Bien lire le fichier `readme.txt` qui s'ouvre à la fin de l'installation, notamment pour les permissions nécessaires. Lire également sa documentation (18).

La fenêtre principale peut ensuite être éditée en XAML pour placer :

- Un sélecteur pour choisir entre la caméra de face et celle de dos
- Une image pour visualiser le flux vidéo
- Un bouton pour prendre une photo

```
<Grid>
     <Grid.ColumnDefinitions>
<ColumnDefinition Width="*"/>
<ColumnDefinition Width="*"/>
<ColumnDefinition Width="*"/>
     </Grid.ColumnDefinitions>
<Grid.RowDefinitions>
<RowDefinition Height="60"/>
<RowDefinition Height="*"/>
<RowDefinition Height="60"/>
</Grid.RowDefinitions>
<Label Grid.Row="0" Grid.Column="0"
     Text="Face" Margin="10"
     HorizontalTextAlignment="End"
     VerticalTextAlignment="Center" />
<Label Grid.Row="0" Grid.Column="2"
     Text="Arrière" Margin="10"
     HorizontalTextAlignment="Start"
     VerticalTextAlignment="Center"/>
<Switch x:Name="type" Grid.Row="0"
     Grid.Column="1"
     HorizontalOptions="Center"/>
<Image x:Name="photo" Grid.Row="1"
     Grid.ColumnSpan="3"
     BackgroundColor="White"
     Margin="20" />
<Button Grid.Row="2" Grid.Column="1"
```



```
      Text="Photo !" Margin="10"
      VerticalOptions="CenterAndExpand"
      />
 </Grid>
```

Capture 58 : définition en XAML de l'écran de capture photo

Dans le projet partagé, dans la fenêtre principale, nous allons placer un attribut qui permettra de choisir la caméra à utiliser.

```
private CameraDevice device;
```

Dans une fonction asynchrone, appelée depuis le constructeur, nous allons commencer par tester si l'appareil peut utiliser ou non le *plugin*, puis initialiser celui-ci et définir les options par défaut :

```
private async Task Init()
{
     if (!CrossMedia.IsSupported)
      {
        await DisplayAlert("Erreur", "Plugin non supporté",
     "Quitter");
        return;
      }
      await CrossMedia.Current.Initialize();
      if(!CrossMedia.Current.IsCameraAvailable)
      {
        await DisplayAlert("Erreur", "Aucune caméra
     disponible", "Quitter");
        return;
      }
      device = CameraDevice.Front ;
}
```

Il faut ensuite choisir la caméra à utiliser quand on déplace le « *switch* ». Pour cela, on utilise l'évènement `toogled` du *switch*:

```
private void type_Toggled(object sender, ToggledEventArgs e)
{
     optionsCamera = new StoreCameraMediaOptions()
      {
        DefaultCamera = e.Value ? CameraDevice.Rear :
     CameraDevice.Front
      };
}
```

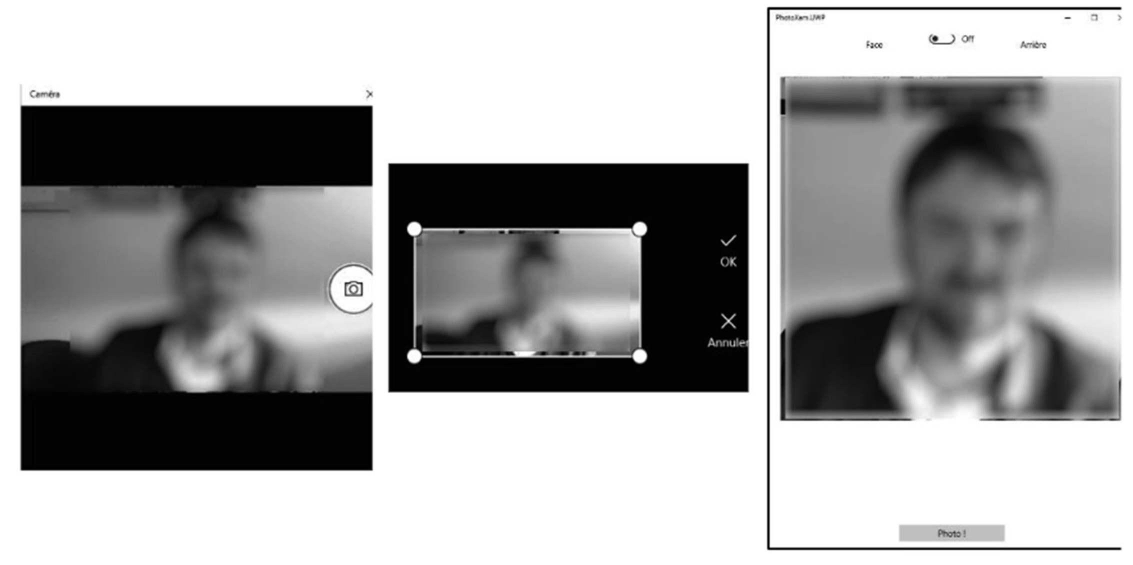
Hélas, ce *plugin* ne permet pas (pour l'instant) de choisir la fenêtre de prévisualisation de la photo : celle par défaut du système sera utilisée.

Pour prendre une photo et la visualiser dans l'image lors de l'appui sur le bouton :

```
private async void Button_Clicked(object sender, EventArgs e)
{
    var file = await CrossMedia.Current.TakePhotoAsync(new
    StoreCameraMediaOptions()
     {
       DefaultCamera = device,
       SaveToAlbum=true
    });
    photo.Source = ImageSource.FromStream(() => {
       var stream = file.GetStream();
       file.Dispose();
       return stream;
    });
}
```

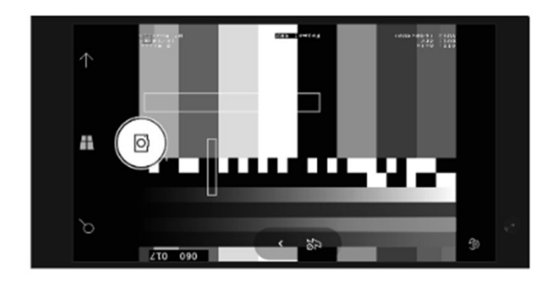
L'image est alors automatiquement sauvée dans l'album du périphérique.

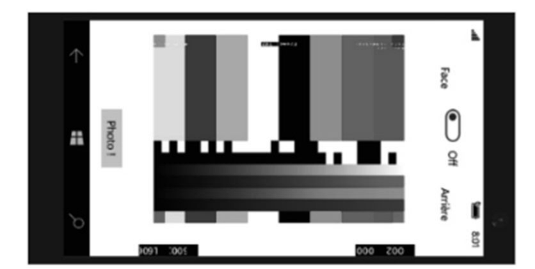
Sous Windows 10, aucune modification particulière à faire, le programme fonctionne directement. Au premier lancement, l'autorisation d'utiliser la caméra sera demandée à l'utilisateur.



Capture 59 : étapes de prise de photo avec Xamarin, Windows 10 sur tablette/PC

Il est également possible d'utiliser un Windows 10 mobile (ou un émulateur W10m, dans ce cas la caméra est simulée par une mire) :





Capture 60 : étapes capture photo avec Xamarin sous émulateur Windows 10 mobile

Avec Android il y a quelques précautions à prendre :

- Le niveau d'API utilisée pour compiler doit être au minimum de 25
- Pour les versions d'Android >= 6.0, il faut ajouter :
    - Une fonction dans le code de l'activité (projet Android)
    - Des informations dans le manifeste
    - Un fichier XML dans les ressources

Si votre application cible les versions les plus récentes d'Android (6.0 ou +) il sera nécessaire d'insérer la fonction suivante dans la classe de l'activité :

```
public override void OnRequestPermissionsResult(int
     requestCode, string[] permissions, Permission[]
     grantResults)
{
     PermissionsImplementation.Current.OnRequestPermissionsRe
     sult(requestCode, permissions, grantResults);
}
```

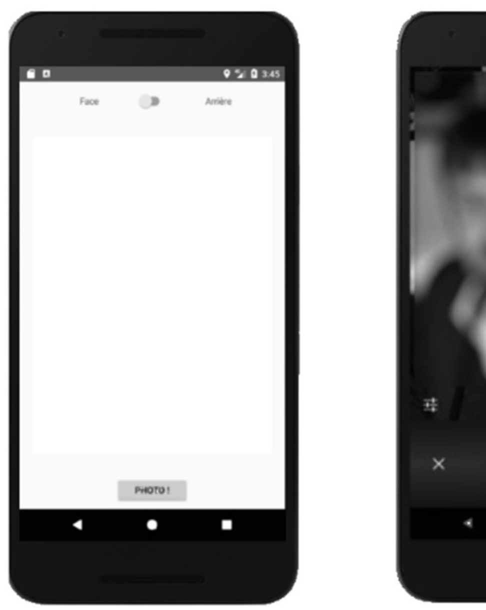
Pour pouvoir sauvegarder les images, les lignes suivantes doivent être ajoutées au fichier du manifeste (modifiez `YOUR_APP_PACKAGE_NAME` pour refléter le nom de votre package) :

```
<provider
     android:name="android.support.v4.content.FileProvider"
     android:authorities="YOUR_APP_PACKAGE_NAME.fileprovider"
     android:exported="false"
     android:grantUriPermissions="true">
     <meta-data
       android:name="android.support.FILE_PROVIDER_PATHS"
        android:resource="@xml/file_paths"></meta-data>
</provider>
```

Il est enfin également nécessaire d'adjoindre le fichier XML suivant aux ressources de votre projet Android afin d'autoriser la sauvegarde dans certains dossiers :

```
<?xml version="1.0" encoding="utf-8"?>
<paths
     xmlns:android="http://schemas.android.com/apk/res/androi
     d">
    <external-files-path name="my_images" path="Pictures"
    <external-files-path name="my_movies" path="Movies" />
</paths>
```

L'émulateur Android étant capable d'utiliser la webcam du PC, il est donc possible de tester le fonctionnement dans celui-ci, mais il est également possible d'utiliser un périphérique réel (téléphone ou tablette).






Capture 61 : étapes capture photo, Xamarin Android

Pour iOS, il faut ajouter dans le fichier `info.plist` un certain nombre de clés pour autoriser l'accès à la caméra et à la bibliothèque d'images.

```
<key>NSCameraUsageDescription</key>
<string>L'application doit accéder à la caméra pour prendre
     des photos</string>
<key>NSPhotoLibraryUsageDescription</key>
<string>L'application doit accéder à la bibliothèque
     d'images</string>
<key>NSMicrophoneUsageDescription</key>
<string>L'application doit accéder au microphone.</string>
<key>NSPhotoLibraryAddUsageDescription</key>
<string>L'application doit accéder à la galerie
     photo.</string>
```

Le simulateur iOS ne possédant pas de caméra, il est impossible de tester l'application avec le simulateur, il faut utiliser un périphérique réel (iPhone, iPod ou iPad).





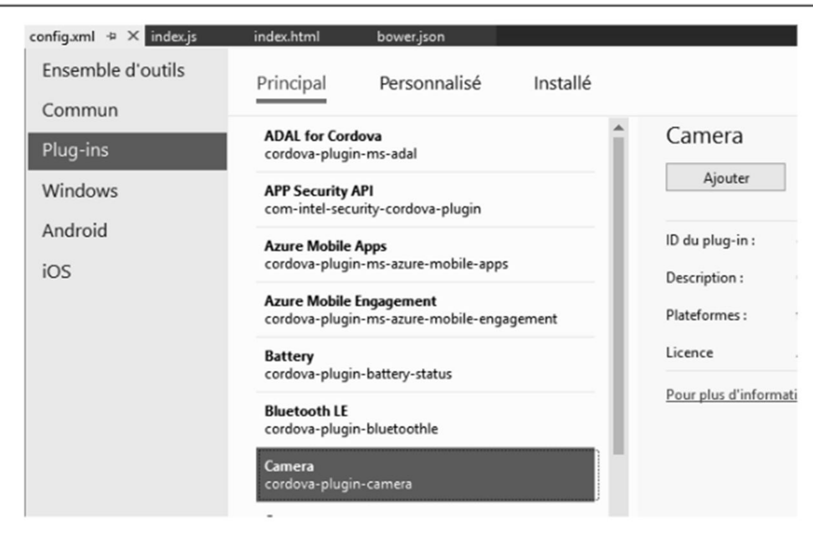
Capture 62 : étapes capture photo, Xamarin iOS

f) Multiplateforme - Cordova

Il faut installer un *plugin* Cordova pour utiliser la caméra. Si vous utilisez les outils Cordova en ligne, il faut exécuter la commande :

```
cordova plugin add cordova-plugin-camera
```

Si vous utilisez Visual Studio, il suffit de double-cliquer sur le fichier `config.xml` du projet, d'ouvrir l'onglet *plug-ins* et de choisir le *plugin* camera.



Capture 63 : installation du plugin Cordova camera sous Visual Studio

Le *plugin* fournit une API pour utiliser la/les caméra(s) (19). Il est donc possible de créer une application multiplateforme en utilisant cette technologie.

La partie IHM de l'application (en HTML donc) comprendra uniquement une zone pour afficher la photo et un bouton pour la capturer, ainsi que la sélection du type de caméra :

```html
<header>
     <div id="choix">
     <select id="typeCamera">
       <option value="Front">Caméra avant</option>
       <option value="Back">Caméra arrière</option>
     </select>
     </div>
     <input type="button" id="clic" value="Photo !" />
</header>        <img id="photo" src="" />
```

La feuille de style associée peut contenir, par exemple :

```css
#photo{
    display:block;
    max-width:90%;
    margin-left:auto;
    margin-right:auto;
}
header{
    text-align:center;
    margin:10px;
}
```

L'initialisation (dans `index.js`) va lier les évènements :

```
var bouton = document.getElementById("clic");
bouton.onclick = function () {
            capturePhoto();
} ;
var choix = document.getElementById("typeCamera");
choix.onchange = function () {
     if (this.selectedIndex == 0)
         choixCamera = Camera.Direction.FRONT;
     else
         choixCamera = Camera.Direction.BACK;
} ;
```

La fonction pour capturer la photo, avec ses fonctions de rappel :

```
function onPhotoOk(imageData) // appelée quand la capture est ok
{
     var image = document.getElementById("photo");
     image.src = "data:image/jpeg;base64,"+imageData;
}

function onPhotoFail(message) // appelée en cas d'echec
{
     console.log(message);
}

function capturePhoto()
{
     if (choixCamera == 0)
       choixCamera = Camera.Direction.FRONT;
     navigator.camera.getPicture(onPhotoOk, onPhotoFail, {
     quality: 50, sourceType:
     Camera.PictureSourceType.CAMERA, destinationType:
     Camera.DestinationType.DATA_URL, cameraDirection:
     choixCamera });
}
```



Capture 64 : Caméra avec Cordova sous Windows, Android, iOS

3. Les autres capteurs

Un *smartphone* peut contenir une quantité d'autres capteurs : boussole, accéléromètre, température, pression, luminosité… Tous les énumérer serait long ; de plus, leur fonctionnement est similaire.

Leur principe est le même pour tous : une fois activés, ils fournissent (à intervalles de temps régulier, ou lors d'un changement, ou à la demande suivant les cas) une valeur captée, souvent sous forme d'un nombre flottant, correspondant à la grandeur physique mesurée.

Suivant l'outil utilisé, un certain nombre de capteurs sont implémentés.

Par convention, le placement du téléphone dans l'espace est défini suivant trois axes x, y, z tels que :

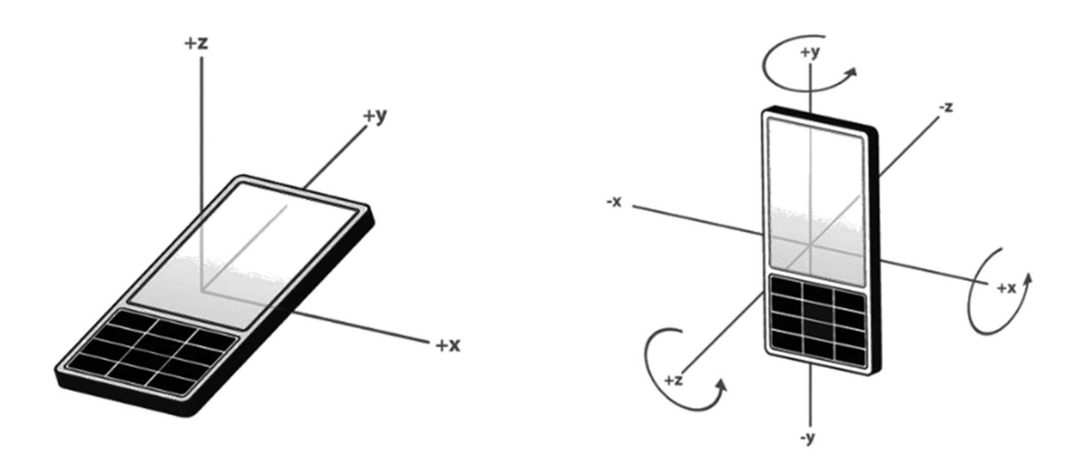


Figure 4 : définition des axes du téléphone

a) Android natif

Tous les capteurs Android fonctionnent sur le même principe. L'API fournit un certain nombre de classes et/ou interfaces pour utiliser les capteurs :

- `SensorManager` est le gestionnaire système de ces capteurs
- `Sensor` représente un capteur
- `SensorEvent` représente un évènement lié à un capteur et contient certains paramètres
- `SensorEventListener` est une interface pour « écouter » un capteur et recevoir ses évènements (voir page 108 pour plus de détails sur les notions d'écouteurs).

Si vous voulez utiliser un capteur, il faut commencer, dans votre activité (en général dans le `onCreate`) par récupérer l'instance du gestionnaire des capteurs :

```
SensorManager manager =
    (SensorManager)getSystemService(SENSOR_SERVICE) ;
```

Une fois ce gestionnaire récupéré, il faut lui demander le capteur relié à un certain type, ici par exemple la boussole :

```
Sensor s =
    manager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD) ;
if( s != null ) // capteur ok…
```

Il existe un grand nombre de capteurs possibles (tous n'étant pas forcément présents sur votre téléphone !) :

- `TYPE_ACCELEROMETER` : accéléromètre ; mesure les secousses, les déplacements du téléphone, en m/s², sur les 3 axes (la gravité est également mesurée puisque c'est une accélération).
- `TYPE_AMBIENT_TEMPERATURE` : température, exprimée en °C.
- `TYPE_GYROSCOPE` : mesure la rotation, en radians par seconde, du périphérique sur les 3 axes. Souvent utilisé dans les jeux…
- `TYPE_ROTATION_VECTOR` : mesure l'orientation, en radians, du périphérique sur les 3 axes.
- `TYPE_LIGHT` : mesure la lumière ambiante, en lux.

- `TYPE_MAGNETIC_FIELD` : mesure le champ magnétique, sur les 3 axes, en µT.
- `TYPE_PRESSURE` : mesure la pression atmosphérique, en hPa.
- `TYPE_PROXIMITY` : mesure la distance d'un objet à l'écran du téléphone. Utilisé pour savoir, par exemple, si une personne porte le téléphone à l'oreille ou non.
- `TYPE_RELATIVE_HUMIDITY` : mesure l'humidité relative de l'air ambiant, en %.

Pour utiliser un capteur dans votre activité, une fois l'instance de celui-ci récupérée, il faut l'écouter ! Pour cela, votre activité doit implémenter `SensorEventListener` et donc redéfinir les opérations suivantes :

- `onSensorChanged` : le capteur a mesuré une nouvelle valeur
- `onAccuracyChanged` : la précision du capteur vient de changer

Il faut inscrire votre activité à l'écoute d'un capteur, en utilisant la méthode `registerListener` du gestionnaire de capteurs. La méthode prend trois paramètres : l'écouteur, le capteur à écouter, la période d'échantillonnage (en µs). Lors de la mesure d'une nouvelle valeur, `onSensorChanged` est donc appelée avec un paramètre du type `SensorEvent`. Cette classe possède 4 propriétés :

- `accuracy` (`int`) : incertitude de mesure
- `timestamp` (`long`) : instant de la mesure, en ns
- `values` (`float[]`) : les valeurs captées (si 3 axes, alors l'ordre est x, y, z).
- `sensor` (`Sensor`) : le capteur qui a déclenché l'évènement.

L'exemple suivant permet d'obtenir l'accélération en m/s² sur les trois axes.

Notre activité ne contient que 3 zones de texte qui afficheront l'accélération. Elle implémente de plus `SensorEventListener`.

```java
public class MainActivity extends AppCompatActivity
     implements SensorEventListener {
         private TextView accelX;
         private TextView accelY;
         private TextView accelZ;
```

Dans le `onCreate`, commençons par initialiser les attributs pour relier le code à la vue :

```
accelX = (TextView)findViewById(R.id.accelX);
accelY = (TextView)findViewById(R.id.accelY);
accelZ = (TextView)findViewById(R.id.accelZ);
```

Il faut ensuite utiliser le `SensorManager` pour se relier au capteur d'accélération :

```
SensorManager manager =
     (SensorManager)getSystemService(SENSOR_SERVICE);
     Sensor accelerometre =
     manager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
     if(accelerometre!=null) {
         manager.registerListener(this,accelerometre,10000);
     }
```

Et enfin, l'opération `onSensorChanged` :

```
@Override
     public void onSensorChanged(SensorEvent sensorEvent) {
         float ax = sensorEvent.values[0];
         float ay = sensorEvent.values[1];
         float az = sensorEvent.values[2];

     accelX.setText(String.valueOf(ax));

     accelY.setText(String.valueOf(ay));

     accelZ.setText(String.valueOf(az));
     }
```

TestSensor
-0.91937345
-0.076614454
9.844957

La capture ci-contre montre un téléphone posé sur une table : on remarque que l'accélération en x et en y est faible, et que l'accélération en z est proche de 9.8 m/s² (gravité terrestre au niveau de la mer).

b) iOS natif

Les différents capteurs n'ont pas la même interface sous iOS avec *Swift* ; chacun d'entre eux utilise des API qui lui sont propres. Dans tous les cas, un type spécifique est défini comme délégué (une implémentation du *pattern* observateur) et le `ViewControler` doit hériter de ce type et s'inscrire

comme « écouteur » du capteur. Une opération est donc ajoutée dans le contrôleur pour récupérer les informations du capteur.

L'exemple suivant permet d'afficher, en temps réel sur l'écran du téléphone, l'angle avec le nord magnétique.

Il faut commencer par faire hériter son `ViewController` de `CLLocationManagerDelegate`, après avoir importé le bon module, et ajouter un attribut du type *ad hoc* :

```
import CoreLocation
class ViewController: UIViewController
    ,CLLocationManagerDelegate {
    let lm = CLLocationManager()
```

Swift

Lors du chargement de la vue (fonction `viewDidLoad`), initialisez la lecture de la boussole :

```
if CLLocationManager.headingAvailable() {
    lm.delegate = self
    lm.startUpdatingHeading()
}
```

Swift

Et enfin, redéfinissez l'opération qui récupère l'angle capté pour, par exemple, l'afficher dans une zone de saisie :

```
func locationManager(manager: CLLocationManager!,
    didUpdateHeading heading: CLHeading!) {
       saisie.txt = String.format(".2f",
    heading.trueHeading)
}
```

Swift

c) Windows natif

Les API pour les capteurs sont regroupés, dans la plateforme UWP, sous l'espace de nom `Windows.Devices.Sensors`.

Il n'y a pas d'API générique, chaque capteur possède sa propre classe mais les interfaces sont très proches :

- `Accelerometer` : accélération
- `Inclinometer` : rotation autour des 3 axes
- `Barometer` : pression atmosphérique
- `Compass` : boussole

- `Gyrometer` : vitesse de rotation autour des 3 axes
- `LightSensor` : luminosité
- `OrientationSensor` : orientation
- `Proximity` : proximité d'un objet avec l'écran
- `Pedometer` : podomètre (mesure du nombre de pas)

Chaque classe possède un évènement `ReadingChanged` pour indiquer qu'une lecture est prête. Chaque classe possède également une méthode statique permettant de récupérer le capteur par défaut du type.

L'exemple suivant donne la valeur de la luminosité ambiante en lux.

Nous allons créer une application Windows universelle dont la page principale ne contient qu'une zone de texte pour afficher, en lux, la luminosité ambiante.

Dans le constructeur, il faut initialiser le capteur (attention : celui-ci peut être `null`, tous les téléphones ne possédant pas ce capteur) et déclarer l'évènement :

```
LightSensor sensor = LightSensor.GetDefault();
if(sensor==null) {
     MessageDialog dial = new MessageDialog("Aucun capteur de
     luminosité", "Erreur");
     dial.ShowAsync();
}
else
     sensor.ReadingChanged += Sensor_ReadingChanged;
```

L'opération de réponse à l'évènement va simplement lire la valeur de luminosité et l'afficher dans la zone de texte, en faisant attention d'utiliser le *thread* UI pour cela :

```
private async void Sensor_ReadingChanged(LightSensor sender,
     LightSensorReadingChangedEventArgs args)
{
     await Dispatcher.RunAsync(CoreDispatcherPriority.Normal,
     () =>
     {
        lux.Text = args.Reading.IlluminanceInLux.ToString();
     });
}
```

d) Les capteurs avec Qt

Qt utilise une API unifiée pour les capteurs, via le module `QSensors`. Les capteurs supportés par plateforme sont définis dans le Tableau 12 (20).

Chaque capteur possède sa propre classe Qt, par exemple `QAccelerometer` pour l'accéléromètre, `QCompass` pour la boussole, mais il est possible d'utiliser la classe de base `QSensor` (ancêtre des autres) en lui indiquant en paramètre le capteur souhaité (une simple chaîne avec le nom de la classe).

`QSensor` possède un certain nombre de signaux qu'il peut envoyer, notamment le plus important : `readingChanged`, qui indique que la valeur captée vient de changer. Les valeurs sont transmises via une instance de `QSensorReading` qui contient les propriétés suivantes :

- `timestamp` : temps, exprimé en µs, de la mesure
- `value(index)` : permet de lire la valeur captée en précisant son index (0=x, 1=y, etc…)
- `valueCount` : précise le nombre de valeurs pouvant être lues avec `value`

Les classes héritées de `QSensor` fournissent une classe héritée de `QSensorReading` dont les propriétés sont plus précises.

L'exemple suivant permet d'indiquer simplement sur l'écran l'orientation (en degrés) sur les 3 axes du téléphone.

Nous commençons par une fenêtre contenant trois `label` pour afficher l'orientation. Un bouton servira à lancer la capture des données.

Dans le projet Qt il faut ajouter le module *ad hoc* :

```
QT       += core gui sensors
```

Nous allons ajouter dans la fenêtre principale un attribut pour le capteur de rotation :

```cpp
    QRotationSensor *sensor;
```

De même, nous rajoutons un *slot* pour la lecture :

```cpp
private slots:
    void lecture();
```

Dans le *slot* relié au bouton, nous allons initialiser le capteur et relier son signal à notre *slot* :

```cpp
sensor = new QRotationSensor(this);
connect(sensor,SIGNAL(readingChanged()),this,SLOT(lecture()))
    ;
```

Capteur			
Accéléromètre	👍	👍	👍
Lumière	👍	👍	👍
Température	👍		
Boussole	👍	👍	👍
Gyroscope	👍	👍	👍
Orientation		👍	👍
Pression	👍		
Proximité	👍	👍	
Rotation	👍	👍	👍

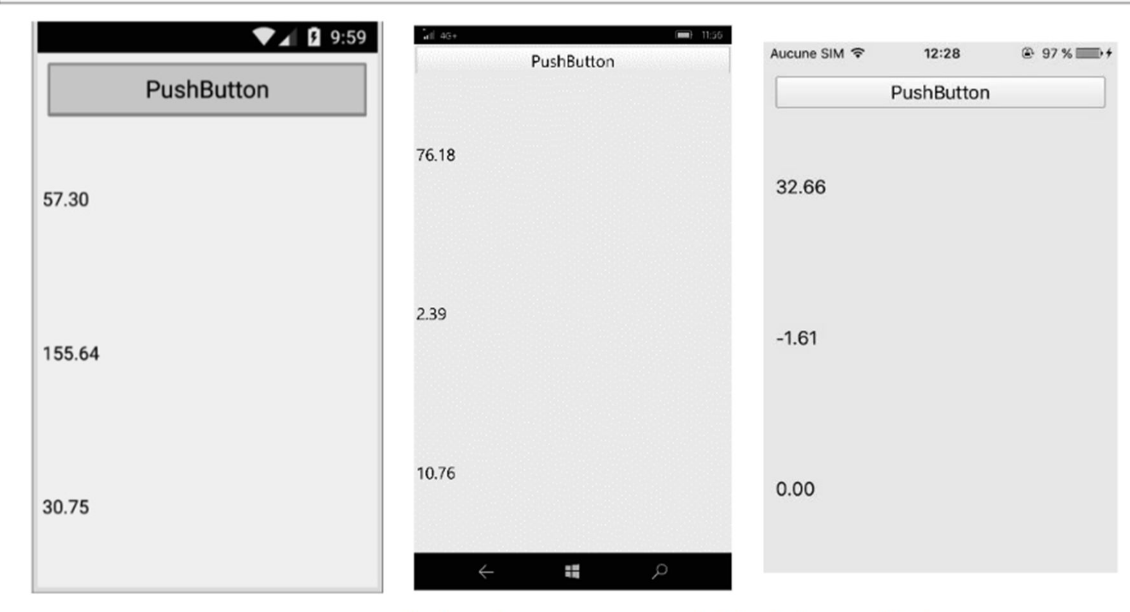
Tableau 12 : compatibilité des capteurs Qt

Il faut ensuite régler sa fréquence d'interrogation (ici par exemple 10 Hz) et lancer la lecture :

```
sensor->setDataRate(10);
if(!sensor->start()) {
     QMessageBox::warning(this,"Erreur","Le capteur ne
     démarre pas");
}
```

Le code du *slot* est très simple et permet de lire les informations :

```
void MainWindow::lecture()
{
    QRotationReading *lecture = sensor->reading();
    float ax = lecture->x();
    float ay = lecture->y();
    float az = lecture->z();
    ui->axeX->setText(QString::number(ax,'f',2));
    ui->axeY->setText(QString::number(ay,'f',2));
    ui->axeZ->setText(QString::number(az,'f',2));
}
```



Capture 65 : capteur d'orientation avec Qt pour Android, Windows mobile, iOS

Il ne faut pas oublier de libérer la mémoire en supprimant le capteur dans le destructeur de la fenêtre :

```
delete sensor ;
```

e) Les capteurs avec Xamarin

Xamarin.Forms ne fournit pas de classes multiplateformes permettant de gérer les capteurs. Il faut donc soit saisir un code différent pour chaque

plateforme (en utilisant par exemple le *pattern* adaptateur (1) pour limiter la taille du code différent), soit utiliser un *plugin*. Il existe de nombreux *plugins Xamarin.Forms* permettant de manipuler les capteurs, notamment :

- `Compass` : gestion de la boussole
- `Device motion` : accéléromètre, gyroscope mais aussi magnétomètre et boussole
- `Sensors` : *plugin* avec de nombreux capteurs (accéléromètre, lumière ambiante, baromètre, boussole, orientation, gyroscope, podomètre, proximité)

Chaque *plugin* est bien entendu différent mais la plupart ont une syntaxe proche.

L'exemple suivant va utiliser le *plugin* `Sensors` pour afficher la valeur du champ magnétique.

La solution *Xamarin.Forms* va comprendre un projet partagé .NET standard qui contiendra, normalement, la totalité du code pour toutes les plateformes.

Il faut commencer par installer le *plugin* à partir de *NuGet* : clic-droit sur la solution, « *gérer les packages NuGet* → *Parcourir* → *Plugin.Sensors* », installer dans tous les projets.

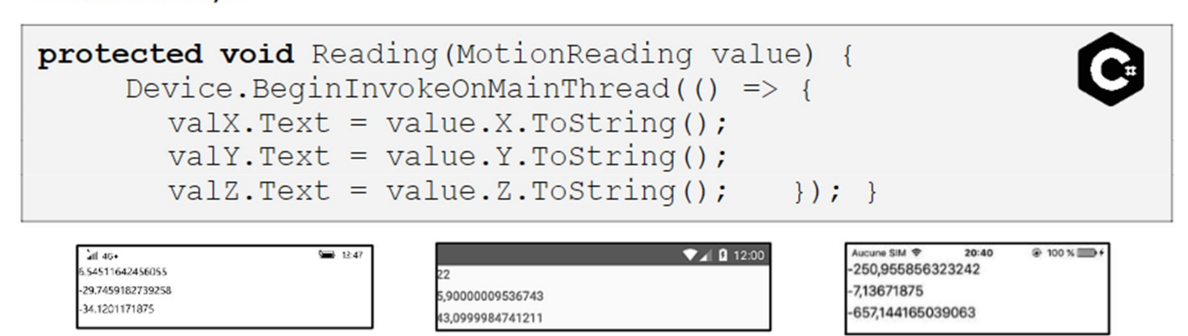
La page principale contient simplement trois `Label` affichant les valeurs du champ magnétique sur les 3 axes.

Dans le constructeur de la page, nous nous assurons que le capteur existe sur la plateforme, puis nous relions une opération de notre classe avec la lecture des données (dans le cas contraire, nous affichons un message d'erreur) :

```
if(CrossSensors.Magnetometer.IsAvailable)
{
     CrossSensors.Magnetometer.WhenReadingTaken().Subscribe(
     Reading);
}
else
{
     DisplayAlert("Erreur", "Aucun magnétomètre", "Ok");
}
```

Il ne reste qu'à créer l'opération de lecture qui met à jour l'affichage (dans le *thread* UI) :

```
protected void Reading(MotionReading value) {
     Device.BeginInvokeOnMainThread(() => {
        valX.Text = value.X.ToString();
        valY.Text = value.Y.ToString();
        valZ.Text = value.Z.ToString();    }); }
```



Capture 66 : magnétomètre avec Xamarin.Forms sous Windows mobile, Android, iOS

f) Les capteurs avec Cordova

HTML5 et Javascript proposent différentes API pour l'utilisation de quelques capteurs (21):

- `DeviceOrientation` : rotation autour des 3 axes
- `DeviceMotion` : gyroscope, accéléromètre
- `AmbientLight` : lumière ambiante
- `Proximity` : distance d'un objet à l'appareil

L'exemple suivant permet d'indiquer sur l'écran la valeur de la rotation lue sur chacun des 3 axes (normalement exprimée en rad/s).

La page HTML comprendra simplement 3 paragraphes vides au départ :

```
<p id="valX"></p> <p id="valY"></p> <p id="valZ"></p>
```

Dans l'opération `onDeviceReady` il suffit d'ajouter un écouteur pour l'évènement `devicemotion` :

```
window.addEventListener("devicemotion", function (event)
     document.getElementById("valX").innerHTML =
     event.rotationRate.alpha;
     document.getElementById("valY").innerHTML =
     event.rotationRate.beta;
     document.getElementById("valZ").innerHTML =
     event.rotationRate.gamma; });
```

Le fonctionnement est similaire avec iOS, Android et Windows.

IX. La gestion du temps

Nous ne parlerons pas ici de météo, mais de la gestion du temps (au sens de l'anglais *time*) dans nos téléphones. En effet, de nombreuses applications ont besoin d'une relation particulière au temps, que ce soit pour exécuter un traitement régulièrement, ou déclencher une action à une heure précise.

1. Exécuter une tâche régulièrement

Dans de nombreuses applications, il est nécessaire d'exécuter une tâche à intervalle de temps régulier, sans bloquer le fonctionnent de l'application. Les systèmes mobiles permettent de faire ceci plus ou moins simplement, suivant le système, l'outil et les API utilisées.

Nous allons dans cette partie explorer ces API en créant, avec chaque outil, une petite application n'affichant qu'un simple compteur à l'écran, avec une durée d'une seconde.

a) Android natif

Java fourni les classes `Timer` (un compteur de temps) et `TimerTask` (une tâche répétitive) pour la gestion des tâches répétées.

Une tâche est une classe qui doit hériter de `TimerTask` et implémenter l'opération `run`. Cette tâche s'exécutera dans un fil d'exécution (*thread*) séparé. Si cette opération doit modifier l'affichage (qui est dans son propre *thread*) il sera nécessaire d'utiliser la fonction `runOnUiThread`.

Pour lancer une tâche répetée, il faut utiliser une instance de la classe `Timer`.

Nous allons observer le fonctionnement des *timers* Android en faisant une application qui affiche un compteur. Cette application ne possède qu'une seule activité dont la disposition ne contiendra qu'un seul texte :

```xml
<TextView android:layout_width="match_parent"
     android:layout_height="wrap_content"
     android:textSize="48sp"    android:id="@+id/counter"
/>
```

Dans le code de l'activité, nous allons ajouter un attribut pour le compteur (un simple entier initialisé à 0) et un attribut pour le contrôle texte :

```
private int compteur=0;
private TextView textView;
```

Cet attribut sera initialisé dans la fonction `onCreate` :

```
textView=(TextView)findViewById(R.id.counter);
```

Il faut également dans cette même classe une opération qui met à jour le compteur :

```
public void majCount()
{
     ++compteur;
     textView.setText(String.valueOf(compteur));
}
```

Dans le même fichier (c'est même possible à l'intérieur de la classe), nous allons créer une classe qui hérite de `TimerTask`, qui est associée avec notre activité et qui, quand elle est exécutée, appelle la mise à jour du compteur (celle-ci interagissant avec la vue doit être exécutée via `runOnUiThread`) :

```
private class MajCount extends TimerTask
     {
          private MainActivity activity;
          public MajCount(MainActivity a) {
              activity =a;
          }
          @Override
          public void run() {
              runOnUiThread(new Runnable(){
                  @Override public void run() {
                      activity.majCount();
                  }});
          }
     }
```

Il faut ensuite utiliser la classe `Timer`, dans l'opération `onCreate`, pour lancer notre tâche répétée (ici toutes les secondes) :

```
Timer timer = new Timer();
timer.schedule(new MajCount(this),0,1000);
```

Lors de l'exécution, toutes les secondes (environ, la précision du `Timer` n'est pas parfaite), l'affichage sera modifié.

b) iOS natif

En *swift* sous iOS, la classe `Timer` permet de réaliser des tâches répétées. La fonction `scheduledTimer` prend comme paramètres l'intervalle de temps (en secondes), l'objet ciblé et l'opération à appeler (le délégué) régulièrement.

Créons notre application affichant un compteur à l'écran. Nous aurons donc dans la vue de l'application un texte (`Label`).

Dans le contrôleur il faut avoir un attribut pour le comptage :

```
private var compteur=0
```

Il faut également une opération qui mette à jour l'affichage :

```
@objc func maj(){
     compteur++
     texte.text = String(compteur)
}
```

Et enfin, dans l'opération de création du contrôleur, lancer le *timer* :

```
var timer = Timer.scheduledTimer(timeInterval: 0.4, target:
     self, selector: #selector(self.maj), userInfo: nil,
     repeats: true)
```

c) Windows natif

Dans le *framework* .NET utilisé par le SDK Windows UWP, il existe une classe `Timer` qui permet de simplement faire exécuter une tâche répétitive.

Lors de sa création, un `Timer` prend en paramètre un délégué, c'est-à-dire une opération qui sera appelée automatiquement. Cette opération est

asynchrone, ne renvoie aucune valeur et prend en paramètre une instance d'`object`. Le paramètre est également passé à la création du `timer`.

L'opération répétitive s'exécutant dans un autre *thread*, elle ne bloque pas l'affichage. En revanche, si celle-ci doit mettre à jour l'affichage, il faut absolument que cette modification ait lieu dans le *thread* UI. Pour cela, il est nécessaire d'appeler la fonction `RunAsync` de l'objet `Dispatcher` de la fenêtre.

Pour étudier le fonctionnement de cette classe, nous allons écrire une petite application qui ne fait que compter et afficher le compteur à l'écran.

Commençons par créer une page XAML qui ne contient qu'une étiquette déposée au centre de l'écran :

```
<TextBlock x:Name="texte" HorizontalAlignment="Center"
    VerticalAlignment="Center" FontSize="96" Text="0"/>
```

Dans la classe correspondante, ajoutons un attribut entier initialisé à 0 :

```
private int compteur = 0;
```

Il faut ensuite écrire une opération asynchrone qui met le compteur à jour :

```
private async void Maj(object state)
{
    ++compteur;
    await Dispatcher.RunAsync(CoreDispatcherPriority.Normal,
       () => {
         texte.Text = compteur.ToString();
       });
}
```

Et enfin, dans le constructeur de la fenêtre, créer un *timer* pour appeler cette fonction régulièrement (ici toutes les 1000 millisecondes, soit 1 seconde) :

```
public MainPage() {
    this.InitializeComponent();
    Timer timer = new Timer(Maj, null, 0, 1000);
}
```

d) Xamarin.Forms

Le *timer* multiplateforme fourni par *Xamarin.Forms* est très simple mais efficace. Il permet d'exécuter, dans un *thread*, un code à intervalle de temps régulier. Ce code étant exécuté au sein du *thread* UI, l'affichage peut être modifié. Pour cela il suffit d'exécuter la fonction `StartTimer` de la classe `Device`. Cette fonction prend deux paramètres : la durée du *timer* et l'opération à exécuter.

L'opération appelée régulièrement ne doit pas prendre de paramètre et doit renvoyer un booléen : si celui-ci est vrai, le *timer* continue, sinon il s'arrête.

Pour tester, réalisons une petite application se contenant d'afficher au centre de l'écran un compteur.

Sur la page XAML principale, il suffit de placer un texte central :

```
<Label x:Name="text" Text="0" FontSize="48"/>
```

Dans la classe correspondante, il faut un attribut pour le comptage :

```
private int compteur = 0;
```

La création du *timer*, très simple, se fait dans le constructeur de la fenêtre (ici avec une durée de 1 seconde) :

```
Device.StartTimer(new TimeSpan(0,0,1), () => {
     ++compteur;
     text.Text = compteur.ToString();
     return true;
});
```

Le fonctionnement est similaire sous Android, iOS et Windows.

e) Cordova

Javascript fournit directement une opération `setInterval` pour exécuter une opération à intervalle de temps régulier. Cette opération prend en paramètre une fonction (n'ayant aucun paramètre et ne renvoyant rien) ainsi que la durée du *timer*, exprimée en millisecondes.

Programmons donc une application affichant un compteur à l'écran.

La vue (le fichier HTML) ne contient qu'un paragraphe de texte :

```
<p id="compteur"></p>
```



Dans la feuille de style CSS les caractéristiques visuelles de ce paragraphe sont :

```
#compteur{
    text-align:center;
    font-size:xx-large;
}
```

Dans le code *Javascript* il est très simple de « lancer » le *timer* pour afficher un compteur dans ce paragraphe :

```
var compteur = 0;
var timer = setInterval(function () {
     document.getElementById("compteur").innerHTML =
     ++compteur;
}, 1000);
```

f) Qt

Le *framework* Qt fournit la classe `QTimer` pour la gestion des tâches répétées.

Cette classe utilise le système des *slots* et signaux Qt. Il suffit pour créer un *timer*, d'avoir un *slot* (une opération ne prenant pas de paramètre et ne renvoyant rien), de le connecter au signal `timeout` du *timer* et de démarrer celui-ci en précisant la durée de l'intervalle (en millisecondes).

Voici maintenant une petite application de test. Commençons par placer une étiquette de texte au centre de l'écran afin d'afficher notre compteur :



Dans le fichier d'entête de la classe associée à la fenêtre (par défaut, `MainWindow.h`), il faut ajouter deux attributs (un entier pour le compteur et le *timer* lui-même) et un *slot* (pour recevoir le signal du *timer*) :

```
private:
    Ui::MainWindow *ui;
    QTimer *timer ;
    int compteur=0;
private slots:
    void maj();
```

Le code du *slot* est le suivant :

```
void MainWindow::maj()
{
    ++compteur;
    ui->texte->setText(QString::number(compteur));
}
```

L'initialisation du *timer* se fait dans le constructeur de la fenêtre :

```
    timer = new QTimer;
    connect(timer,SIGNAL(timeout()),this,SLOT(maj()));
    timer->start(1000);
```

2. Programmer une alarme

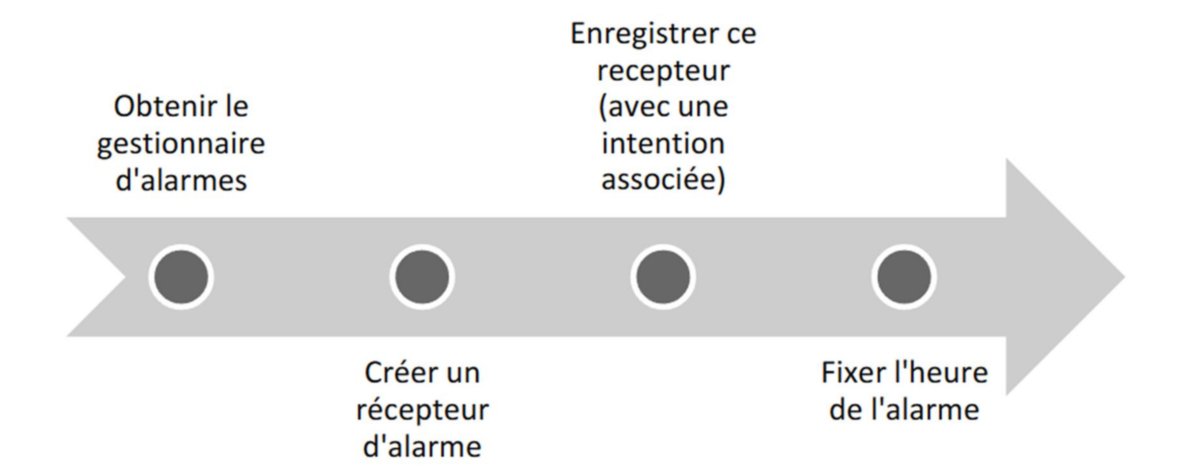
Une alarme permet de faire exécuter une opération à un instant précis (en général une heure précise). Dans certains cas, l'alarme est capable de « réveiller » le téléphone (sortie du mode veille). L'alarme, une fois lancée, ne nécessite pas que l'application soit active.

a) Android natif

Android utilise plusieurs classes pour la gestion des alarmes :

- `AlarmManager` : le gestionnaire des alarmes
- `BroadcastReceiver` : le récepteur de l'alarme (la fonction appelée au terme de l'alarme)
- `Intent` et `PendingIntent` (l'action à effectuer)

Il faut, pour lancer l'alarme, un certain nombre d'étapes :



Pour obtenir le gestionnaire de l'alarme, il suffit de le demander au système :

```
AlarmManager am =
    (AlarmManager)getSystemService(ALARM_SERVICE);
```

Pour créer un récepteur d'alarme, il est nécessaire d'écrire une classe qui hérite de `BroadcastReceiver` et qui implémente son opération `onReceive`.

Par exemple, le récepteur suivant affiche un simple `Toast` (un message texte fugitif) :

```
public class Alarme extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent){
   Toast.makeText(context,"Debout",Toast.LENGTH_LONG).show();
  }
}
```

Pour enregistrer le récepteur auprès du gestionnaire d'alarmes, nous devons choisir un nom le représentant :

```
AlarmManager am = (AlarmManager)
     getSystemService(ALARM_SERVICE);
BroadcastReceiver rec = new Alarme();
     registerReceiver(rec,new IntentFilter("MonAlarme"));
```

Pour fixer l'heure de l'alarme, la première étape est de créer un objet de type `PendingIntent` associé avec une intention correspondant au nom du récepteur, puis d'appeler la fonction `set` du gestionnaire d'alarmes avec comme paramètres :

- Le mode de l'alarme :
  - `ELAPSED_REALTIME` : temps écoulé depuis le démarrage du système
  - `ELAPSED_REALTIME_WAKEUP` : idem mais l'alarme est capable de réveiller le téléphone
  - `RTC` : heure exprimée en temps universel
  - `RTC_WAKEUP` : idem mais l'alarme est capable de réveiller le téléphone
- Le délai de l'alarme (en millisecondes, dépend du mode précédent)
- L'objet de type `PendingIntent` précédemment créé

Le code suivant déclenchera l'alarme (avec le récepteur préalablement créé) 30 secondes après son exécution :

```
PendingIntent pi = PendingIntent.getBroadcast(this,0,new
    Intent("MonAlarme"),0);
am.set(AlarmManager.RTC,System.currentTimeMillis()+30000,pi);
```

b) iOS natif

Les API iOS ne distinguent pas les alarmes des notifications locales différées : reportez-vous page 223.

c) Windows natif

Les API Windows UWP ne distinguent pas les alarmes des notifications locales différées : reportez-vous page 222.

d) Xamarin.Forms

Xamarin.Forms ne propose pas d'outil multiplateforme spécifique pour faire une alarme. Il est possible d'utiliser un *timer* « mono coup » (voir page 213).

e) Qt

Qt ne contient rien, dans son *framework*, pouvant être utilisé pour faire une alarme, à part la classe `QTimer` déjà abordée page 214, mais elle n'est pas capable de « réveiller » l'appareil.

f) Cordova

Il n'y a rien, en Javascript, permettant de réveiller le périphérique, seule la fonction `setTimeout` permet d'exécuter une tâche à un temps donné, mais si le périphérique est en veille, elle n'aura pas lieu.

Un *plugin* existe pour permettre un « réveil » ; ce *plugin* ne fonctionne que sous Android et iOS (non Windows donc). Il ne fait pas partie des *plugins* standard de Cordova, il faut l'installer via son chemin sur GitHub[1].

Principal Personnalisé Installé

Sélectionner une méthode de récupération du plug-in

ID du plug-in Local Git

Entrer un emplacement de référentiel Git pour le plug-in à installer

https://github.com/wnyc/cordova-plugin-wakeuptimer.git →

Capture 67 : installation du plugin Cordova pour alarme

Une fois le *plugin* installé, il est très simple de demander le lancement d'une alarme à une heure précise.

Il faut commencer par écrire une fonction qui sera appelée lors de l'alarme. La fonction prend un paramètre qui est le résultat : il possède une propriété `type` qui peut être, entre autres, « *wakeup* » (l'alarme est arrivée à terme). Par exemple, la fonction pourrait afficher une simple alerte :

```
function success(resultat) {
    if (resultat.type == "wakeup")
        alert("wakeup");
}
```

[1] Plateforme de partage de logiciels *opensource*, utilisant le protocole Git.

De même, créons une fonction à appeler en cas d'erreur :

```
function error() {
    alert("erreur");
}
```

Pour déclencher l'alarme, il suffit d'appeler la fonction `wakeup` du *plugin* en lui transmettant les deux fonctions écrites (celle à appeler en cas de succès, et celle à exécuter en cas d'erreur), ainsi que les différentes alarmes via un tableau JSON :

```
window.wakeuptimer.wakeup(success, error, {
    alarms: [{
        type: 'onetime',
        time: { hour: 21, minute: 35 },
        extra: { message: "test" },
        message: "C'est l'heure !"
    }]
});
```

Cette fonction ne doit être appelée qu'une fois que le périphérique est prêt, donc au plus tôt à la fin de la fonction `onDeviceReady`. Bien entendu, vous pouvez l'appeler après !

3. Notifications locales

Une notification est un moyen de prévenir l'utilisateur. Sur la plupart des téléphones, la notification apparait en haut du téléphone et prévient l'utilisateur (par un son, une vibration, ou autre) d'un évènement quelconque.

a) Android natif

Les notifications Android sont assez complètes ; elles sont cependant exécutées directement. Si l'on souhaite différer une notification, il faut la lancer dans une alarme (voir page 215).

Le mécanisme des notifications Android nécessite les classes suivantes :

- `NotificationManager` : le gestionnaire de notifications
- `Notification` : la notification elle-même
- `Notification.Builder` : l'objet créateur de notifications
- `Intent` et `PendingIntent` : l'activité à utiliser

Une notification comporte :

- Une icône (obligatoire) utilisée pour l'affichage dans la barre d'état du téléphone
- Un titre (obligatoire) qui sera affiché à l'utilisateur
- Une description indiquant l'utilité de la notification
- Un ou plusieurs signal(aux) pour attirer l'attention de l'utilisateur (facultatif) : son, vibration, clignotement du flash…

Une notification est associée à une activité : elle sera exécutée quand l'utilisateur aura cliqué sur la notification.

Pour étudier le fonctionnement, nous allons réaliser une petite notification simple.

Il faut tout d'abord un fichier image dans les ressources, de petite taille, pour l'icône de la notification. Puis une activité principale, où un simple bouton lancera la notification, et enfin une activité exécutée par la notification.

La disposition de l'activité principale ne contient qu'un simple bouton :

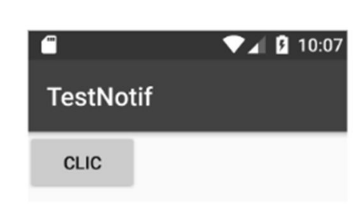
```
<Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Clic"
        android:onClick="clic"
        />
```

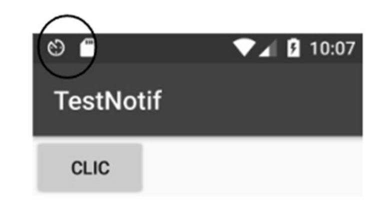
La fonction de réponse au clic du bouton dans l'activité principale sert à créer la notification :

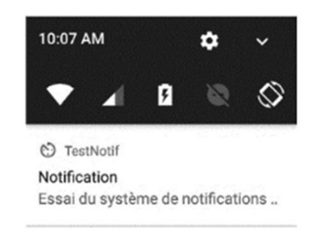
```
@RequiresApi(api = Build.VERSION_CODES.JELLY_BEAN)
public void clic(View v) {
    NotificationManager manager =
(NotificationManager)getSystemService(NOTIFICATION_SERVICE);
    Intent intent = new Intent(this,NotifActivity.class);
    PendingIntent pi =
```

```
PendingIntent.getActivity(this,0,intent,
PendingIntent.FLAG_ONE_SHOT);
    Notification.Builder builder = new
Notification.Builder(this);
    builder.setWhen(System.currentTimeMillis());
    builder.setTicker("Notification");
    builder.setContentTitle("Notification");
    builder.setContentText("Essai du système de notifications
Android");
    builder.setSmallIcon(R.drawable.ic_stat_name);
    manager.notify(1,builder.build());
}
```

L'exécution provoque les écrans suivants :







Capture 68 : détails d'une notification Android

La notification peut être enrichie avant d'être lancée, en ajoutant des opérations lors de sa construction.

Le plus simple est de récupérer le son standard de notification :

```
Uri sound = RingtoneManager.getDefaultUri(
     RingtoneManager.TYPE_NOTIFICATION);
builder.setSound(sound) ;
```

Pour faire vibrer le téléphone on précise un motif de vibration (durée de vibration, durée d'arrêt, etc.). Le motif comprend un multiple de 3 valeurs : la première est le délai avant vibration, la 2e la durée (en ms) de la vibration, la 3e la durée de l'attente.

```
long[] motif = new long[] {0,1000,500} ; // vibre 1s, attend 0.5s
builder.setVibrate(motif) ;
```

Il faut déclarer dans le manifeste l'autorisation d'utiliser le vibreur :

```
<uses-permission android:name="android.permission.VIBRATE"/>
```

Il est également possible de faire « *flasher* » la LED du téléphone (si disponible). La fonction utilise 3 paramètres : la couleur à utiliser (sur 32 bits

ARVB[1]), la durée de l'allumage et la durée d'extinction (le cycle est répété tant que la notification n'est pas prise en compte). Exemple pour une *led* blanche avec un clignotement 500ms allumée / 250 ms éteinte :

```
builder.setLights(0xFFFFFFFF,500,250);
```



b) Windows natif

Les API Windows appellent les notifications « `Toast` ». L'application doit demander l'autorisation pour les utiliser. Il faut pour cela éditer les propriétés du projet et choisir dans l'onglet « Application » les notifications de l'écran de verrouillage sur « badge et texte de vignette ». Il faut évidemment une image pour le badge, déclarée dans l'onglet « actifs visuels ».

Notification de l'écran de verrouillage : Badge et texte de vignette

Trois classes sont fournies :

- `ToastNotification` : la notification elle-même
- `ToastNotificationManager` : le gestionnaire des notifications
- `ToastNotifier` : l'objet qui lance la notification

Le contenu d'une notification est très complet et peut être personnalisé. Il faut un document XML pour le définir. Certains modèles existent pour simplifier le travail. L'exemple suivant fonctionne pour une notification avec un titre et une zone de texte :

```
var content =
    ToastNotificationManager.GetTemplateContent(ToastTemplat
    eType.ToastText01);
var lines = content.GetElementsByTagName("text");
lines[0].AppendChild(content.CreateTextNode("Ceci est une
    notification !"));
```

La notification peut être programmée à une date précise ou être lancée directement. Pour cela, il faut créer un objet :

[1] Couleur ARVB : couleur exprimée sous forme d'un entier 32 bits. Les 8 bits de poids fort représentent le canal alpha (transparence), les 8 bits suivant la composante rouge, les 8 suivants la composante verte et les 8 bits de poids faible la composante bleue. La valeur 0xFFFFFFFF représente un blanc, par exemple.

- De type `ScheduledToastNotification` pour une notification différée
- De type `ToastNotification` pour un lancement immédiat

Exemple pour créer une notification à partir du contenu précédent à une date (type `DateTime`) donnée :

```
ScheduledToastNotification toast = new
     ScheduledToastNotification(
     content, new DateTimeOffset(date));
toast.Id = APP_ID; // identifiant de type chaîne
```

Pour « lancer » la notification, il faut passer par un `ToastNotifier` :

```
ToastNotifier notifier =
     ToastNotificationManager.CreateToastNotifier();
notifier.AddToSchedule(toast);
```

c) iOS natif

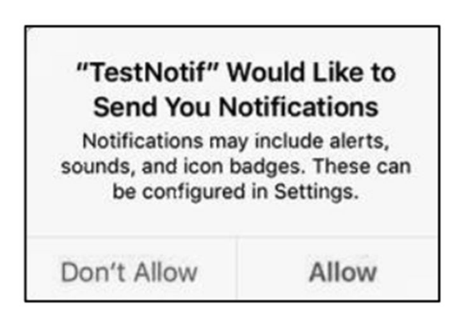
Il existe 2 API de notifications en *Swift* : une plus ancienne, antérieure à iOS 10 et donc dépréciée, et l'autre plus récente.

Nous allons tout d'abord présenter la première API, puis la plus récente. Adaptez l'API utilisée à la version de iOS ciblée !

L'application doit être autorisée à utiliser les notifications. Il faut donc le faire en ajoutant le code suivant dans `AppDelegate.swift`, à la fin de la fonction `application` :

```
application.registerUserNotificationSettings(UIUserNotificati
     onSettings(types :[.alert,.badge,.sound],categories :nil
     ))
```

Lors de la première exécution, l'utilisateur devra confirmer l'autorisation :

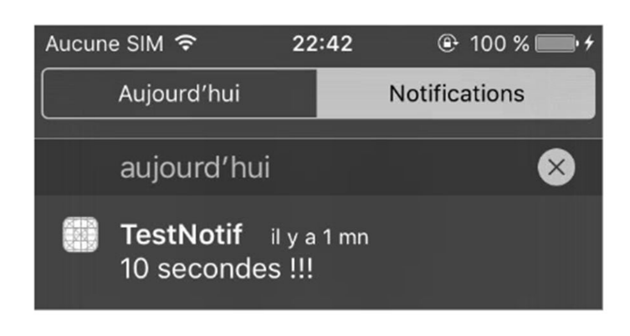


Il faut tout d'abord une date/heure à laquelle lancer la notification. Cette date peut être définie intégralement, par un `UIDatePicker` ou relativement (voir les constructeurs de la classe `Date`). Par exemple, la date suivante est située 75 secondes dans le futur :

```
let laDate = Date(timeIntervalSinceNow : 75)
```

Il faut ensuite construire la notification, choisir sa date de lancement, le message à afficher, etc. Le code ci-dessous lance une simple notification, 10 secondes après l'appel :

```
let notif = UILocalNotification()
notif.fireDate = Date(timeIntervalSinceNow :10)
notif.alertBody = "10 secondes !!!"
notif.soundName = UILocalNotificationDefaultSoundName
notif.alertAction = "ouvrir"
UIApplication.shared.scheduleLocalNotification(notif)
```





Capture 69 : Notifications iOS en mode normal, en veille

Depuis iOS 10, les notifications précédentes sont obsolètes et sont remplacées par un autre système (qui est assez proche). Pour l'utiliser, commencer par importer le nouveau *framework* dans le code du contrôleur (fichier `viewcontroller.swift`)

```
import UserNotifications
```

Il est nécessaire, comme pour l'ancien système, demander l'autorisation à l'utilisateur, ici à la fin du chargement (fonction `viewDidLoad`) :

```
UNUserNotificationCenter.current().requestAuthorization(optio
    ns :[.alert,.sound,.badge], completionHandler :nil)
```

Créons ensuite la notification :

```
let notif = UNMutableNotificationContent()
notif.title = "Le titre"
notif.subtitle = "Le sous-titre !"
notif.body = "Le texte de la notification…"
notif.categoryIdentifier = "message"
```

Le déclencheur de la notification peut être un temps (comme la précédente), mais aussi une localisation particulière. Ici nous créons un déclencheur après intervalle de temps :

```
let trigger =
    UNTimerIntervalNotificationTrigger(timeInterval :30,
    repeats :false)
```

Avant de lancer la notification, il suffit de créer une requête d'exécution :

```
let request = UNNotificationRequest(identifier="Mon.ID",
    content :notif, trigger :trigger)
```

L'identifiant sert à donner à chaque notification un code qui est normalement unique. Il ne reste plus qu'à ajouter la notification au centre de gestion :

```
UNUserNotificationCenter.current().add(request,
    withCompletionHandler :nil)
```

Le fonctionnement est similaire au code vu plus haut.

d) Xamarin.Forms

Comme tout API un peu spécifique à une plateforme il n'existe pas de solution standard en *Xamarin.Forms* pour gérer les notifications locales. Il existe heureusement un *plugin* (`Xam.Plugins.Notifier`) qui s'en charge sur toutes les plateformes.

Il faut donc installer ce package *NuGet* via le gestionnaire de package pour la solution.

Le package ne contient qu'une seule classe, `CrossLocalNotification`. Cette classe ne possède qu'une seule opération, `Show`, qui permet de lancer la notification immédiatement ou à une heure donnée.

La notification est uniquement constituée d'un titre et d'un texte.

Pour Android, il est possible de régler l'icône de la notification (dans les autres cas c'est l'icône de l'application qui est utilisée) :

```
LocalNotificationsImplementation.NotificationIconId =
      Resource.Drawable.icon;
```

Le lancement de la notification se fait tout simplement :

```
CrossLocalNotifications.Current.Show("Titre", "Le texte de la
     notification");
```

Si l'on souhaite différer la notification il suffit d'ajouter l'heure de délivrance de celle-ci (dans l'exemple suivant, 10s après) :

```
CrossLocalNotifications.Current.Show("Titre", "Notification
     décalée de 10s", 101, DateTime.Now.AddSeconds(10));
```

Certaines permissions sont nécessaires pour certaines plateformes. Inutile pour Android, mais il faut autoriser les « toast » pour Windows et les notifications locales pour iOS.

Pour Windows, ajoutez un logo pour le `Badge`, et autorisez dans le manifeste :

Notification de l'écran de verrouillage : Badge et texte de vignette

Pour iOS, il faut créer une classe telle que la suivante (inutile sur iOS < 10) :

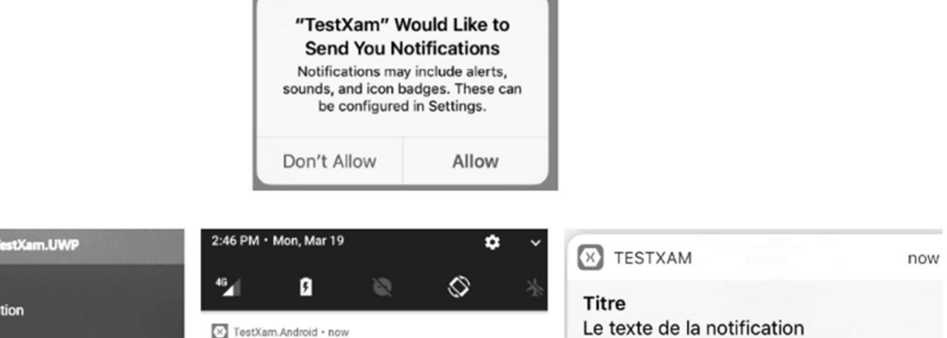
```
public class UserNotificationCenterDelegate :
     UNUserNotificationCenterDelegate
{
     public override void
     WillPresentNotification(UNUserNotificationCenter center,
     UNNotification notification,
     Action<UNNotificationPresentationOptions>
     completionHandler) {
completionHandler(UNNotificationPresentationOptions.Alert);
     }
}
```

Il est également obligatoire d'ajouter, dans l'opération `FinishedLaunching` de la classe `AppDelegate`, le code suivant pour demander l'autorisation de notifier l'utilisateur :

```
if(UIDevice.CurrentDevice.CheckSystemVersion(10, 0))
{
UNUserNotificationCenter.Current.RequestAuthorization(
     UNAuthorizationOptions.Alert |
     UNAuthorizationOptions.Badge |
     UNAuthorizationOptions.Sound,
 (approved, error) => { });

UNUserNotificationCenter.Current.Delegate = new
     UserNotificationCenterDelegate();
}
else if (UIDevice.CurrentDevice.CheckSystemVersion(8, 0))
{
     var settings =
     UIUserNotificationSettings.GetSettingsForTypes(
      UIUserNotificationType.Alert |
     UIUserNotificationType.Badge |
     UIUserNotificationType.Sound, new NSSet());
     UIApplication.SharedApplication.RegisterUserNotification
     Settings(settings);
}
```

Lors de la première exécution, l'application demandera à l'utilisateur :



Capture 70 : notification avec Xamarin.Forms sous Windows, Android, iOS

Il est possible de ne pas utiliser de *plugin* : il est alors nécessaire d'implémenter une classe par plateforme (Android, iOS, Windows) en utilisant les outils natifs.

Si la notification doit être utilisée dans le projet partagé, c'est possible : il faut alors définir dans ce projet une interface :

```
public interface INotifier {
     void Envoyer(string titre, string message, DateTime
     when);
}
```

Pour appeler la notification, le plus simple est d'utiliser une classe singleton[1] qui sera associée avec l'interface précédente :

```
public class GereNotif {
     private static GereNotif instance = null;
     private GereNotif() { }
     public static GereNotif Instance     {
        get
        {
           if (instance == null)
              instance = new GereNotif();
           return instance;
        }
     }

     public INotifier Notifier
     { get => notifier; set => notifier = value; }

     private INotifier notifier;

     public void Envoyer(string titre, string message,
     DateTime time)
     {
     if (notifier != null)
        notifier.Envoyer(titre, message, time);
     }
}
```

Il faut ensuite, dans chaque projet spécifique à une plateforme, créer une classe qui implémente l'interface `INotifier`.

La classe est la suivante en UWP :

```
class UWPNotifier : INotifier {
     public void Envoyer(string titre, string message,
```

[1] Un singleton est une classe ne possédant qu'une seule instance. C'est un patron de conception (*design pattern*) classique.

```
    DateTime when)   {
      var content =
    ToastNotificationManager.GetTemplateContent(ToastTemplat
    eType.ToastImageAndText01);
      var lines = content.GetElementsByTagName("text");
      lines[0].AppendChild(content.CreateTextNode(message));
      ScheduledToastNotification toast = new
    ScheduledToastNotification(content, new
    DateTimeOffset(when));
      toast.Id = titre;
      var notifier =
    ToastNotificationManager.CreateToastNotifier();
    notifier.AddToSchedule(toast);
    }
}
```

En Android, la classe est un peu plus complexe :

```
class AndroidNotifier : BroadcastReceiver, INotifier
{
    private Context context;
    private string titre;
    private string message;
    public AndroidNotifier(Context context)
    {
      this.context = context;
    }
    public void Envoyer(string titre, string message,
    DateTime when)
    {
      this.titre = titre;
      this.message = message;
      AlarmManager am =
    context.GetSystemService(Context.AlarmService) as
    AlarmManager;
      context.RegisterReceiver(this, new
    IntentFilter("Alarme"));
      PendingIntent pi = PendingIntent.GetBroadcast(context,
    0, new Intent("Alarme"), 0);
      DateTime baseTime = new DateTime(1970, 1, 1, 0, 0, 0,
    DateTimeKind.Utc);
      TimeSpan span = when.ToUniversalTime() - baseTime;
    am.Set(AlarmType.RtcWakeup,
    (long)span.TotalMilliseconds, pi);
    }

    public override void OnReceive(Context context, Intent
    intent)
```

```
    {
      NotificationManager manager =
    context.GetSystemService(Context.NotificationService) as
    NotificationManager;
      PendingIntent pi = PendingIntent.GetActivity(context,
    0, new Intent(), PendingIntentFlags.OneShot);
      Notification.Builder builder = new
    Notification.Builder(context);
      builder.SetTicker("Notif");
      builder.SetContentTitle(titre);
      builder.SetContentText(message);
      builder.SetSmallIcon(Resource.Drawable.badge);
      manager.Notify(100, builder.Build());
    }
}
```

Enfin, pour iOS, la classe est la suivante (elle comprend du code pour demander l'autorisation d'afficher une notification) :

```
class IOSNotifier : INotifier {
    public IOSNotifier(UIApplication app)     {
      if(UIDevice.CurrentDevice.CheckSystemVersion(8,0))
      {
        var settings =
    UIUserNotificationSettings.GetSettingsForTypes(UIUserNot
    ificationType.Alert | UIUserNotificationType.Badge |
    UIUserNotificationType.Sound,null);
        app.RegisterUserNotificationSettings(settings);
      }
    }
    public void Envoyer(string titre, string message,
    DateTime when)   {
      UILocalNotification notification = new
    UILocalNotification
      {
        FireDate = NSDate.FromTimeIntervalSinceNow( (when-
    DateTime.Now).TotalSeconds),
        AlertAction = titre,
        AlertBody = message,
        ApplicationIconBadgeNumber=1,
        TimeZone=NSTimeZone.DefaultTimeZone,
        SoundName = UILocalNotification.DefaultSoundName
      };
      UIApplication.SharedApplication.ScheduleLocalNotificat
    ion( notification);
      }
}
```

Pour iOS il faut également ajouter, dans la classe `AppDelegate` l'opération suivante (le code de l'alerte peut changer) :

```
public override void ReceivedLocalNotification(UIApplication
     application, UILocalNotification notification) {
     UIAlertController okayAlertController =
     UIAlertController.Create(notification.AlertAction,
     notification.AlertBody, UIAlertControllerStyle.Alert);
     okayAlertController.AddAction(UIAlertAction.Create("OK",
     UIAlertActionStyle.Default, null));
     UIApplication.SharedApplication.KeyWindow.RootViewContro
     ller.PresentViewController(okayAlertController, true,
     null);
     UIApplication.SharedApplication.ApplicationIconBadgeNumb
     er = 0;
}
```

Il ne reste plus qu'à initialiser, dans le constructeur de l'écran principal de chaque plateforme, le lien entre le singleton et un `notifier`.

En UWP, à la fin du constructeur de la `MainPage` :

```
GereNotif.Instance.Notifier = new UWPNotifier();
```

En Android, à la fin de l'opération `OnCreate` de la `MainActivity` :

```
GereNotif.Instance.Notifier = new AndroidNotifier(this);
```

En iOS, à la fin de l'opération `FinishedLaunching` de la classe `AppDelegate` :

```
GereNotif.Instance.Notifier = new IOSNotifier(app);
```

e) Qt

Qt ne propose pas, à ce jour, d'API particulière pour gérer de manière multiplateforme les notifications. Il est possible d'utiliser les API natives du système, mais en perdant le côté multiplateforme. L'exemple `Qt Notifier` (http://doc.qt.io/qt-5/qtandroidextras-notification-example.html) montre comment appeler du code Android/Java natif depuis une application Qt.

f) Cordova

Cordova doit utiliser un *plugin* pour la gestion des notifications locales, ou la nouvelle API Javascript « Notification ».

Le *plugin* standard `cordova-plugin-dialogs` est appelé « notifications » mais il ne sert qu'à afficher des dialogues standards (type `MessageBox`) : ce ne sont pas des « vraies » notifications.

Le *plugin* de katzer (https://github.com/katzer/cordova-plugin-local-notifications.git) quant à lui, permet bien de traiter les notifications pour iOS (>=10), Windows (>=10) et Android (>=4.4). Il ne fonctionnera hélas pas sur les systèmes plus anciens que ceux cités.

Installez-le : dans Visual Studio, double-cliquez sur `config.xml`, menu *plug-ins*, *Personnalisé, git*, et entrez l'URL du *plugin*.

Pour simplement créer une notification différée :

```
cordova.plugins.notification.local.schedule({
                title: "Titre de la notif",
                text: "Contenu de la notification",
                foreground: true,
                id: 1,
                trigger: { in: 10, unit: "second" }
            });
```

Il y a de nombreux réglages possibles, consultez la documentation du *plugin* pour plus de renseignements.

X.Les services web

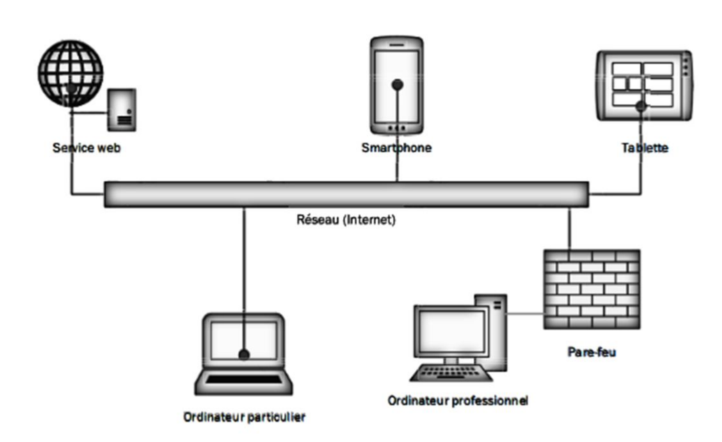
1. Définition

Un **service web** est un programme, fourni par un serveur relié à Internet (ou à un Intranet) qui permet à un agent distant (un autre ordinateur relié au même réseau) d'utiliser ses ressources. Un tel programme permet donc de réaliser des applications distribuées (réparties sur plusieurs machines) en utilisant les protocoles du web.

Le service est dit « web » car il utilise le protocole réseau *http* pour le transport des données, comme les sites web. De ce fait, il traverse les pare-feux d'entreprises qui sont souvent configurés pour ne laisser passer que les services classiques, comme le web, sur le port 80.

Un tel logiciel requiert des protocoles standardisés pour l'échange des données ; celles-ci se faisant au format texte, en général XML ou JSON, il n'y a pas besoin d'avoir une plateforme particulière pour utiliser le service web. Le client (ou consommateur) du service peut donc être n'importe quel agent logiciel qui peut nécessiter les protocoles du web : un site web, une application bureau ou, bien évidemment, une application sur mobile.

On peut considérer que le service web fournit l'implémentation de certaines méthodes pouvant être appelées à distance, au travers d'un réseau TCP/IP (Internet ou Intranet) et transportées par *http* dans un format textuel.



On accède à un service web avec une URL, comme toute ressource web, et une requête *http* (type *GET* ou *POST*). Il répond par une réponse *http* comme le prévoit ce protocole (22).

Il existe de nombreux services web publics, certains gratuits et d'autres payants avec un abonnement ou au nombre de consultations. Il est possible de créer son propre service web, mais il faut dans ce cas un hébergement web.

2. Utilisation d'un service web

De nombreuses applications mobiles utilisent des services web. Bien entendu, il faut que l'appareil soit relié à Internet, via une connexion 4G ou Wifi. Dans de nombreux cas, nos applications feront appel à un service web.

a) Utilisation d'une donnée externe

Il arrive qu'une application ait besoin d'une donnée disponible sur un serveur web : la météo du jour par exemple, comme dans l'atelier page 389, ou le cours d'une devise. Dans ce cas, notre application va consommer le service web pour obtenir la donnée voulue. Il serait en effet peu pratique que ce type de données soit inclus dans l'application : il faudrait la mettre à jour tous les jours !

b) Base de données importante

Même si la quantité mémoire des téléphones a beaucoup augmenté, elle reste faible et il est impossible de stocker de grosses bases de données sur un téléphone ! Si de plus ces données sont partagées, elles doivent être placées sur un SGBD[1] accessible par le réseau. Or il est impossible d'accéder directement au SGBD via le téléphone, pour plusieurs raisons :

- Si le fournisseur de données désire changer de SGBD, il faudrait mettre à jour les applications…
- Le SGBD est fréquemment derrière un pare-feu donc inaccessible directement
- Le SGBD n'est pas obligatoirement relié à Internet

[1] Système de Gestion de Bases de Données : logiciel de type serveur (par exemple : MySQL) permettant de stocker, retrouver, mettre à jour de grandes quantités de données (45).

- Le système du téléphone ne possède pas de connecteur pour tous les SGBD existants

Dans ce cas, nous utiliserons un modèle en trois niveaux : un service web permettra de fournir les données qu'il ira chercher dans le SGBD, comme sur le schéma ci-dessous :

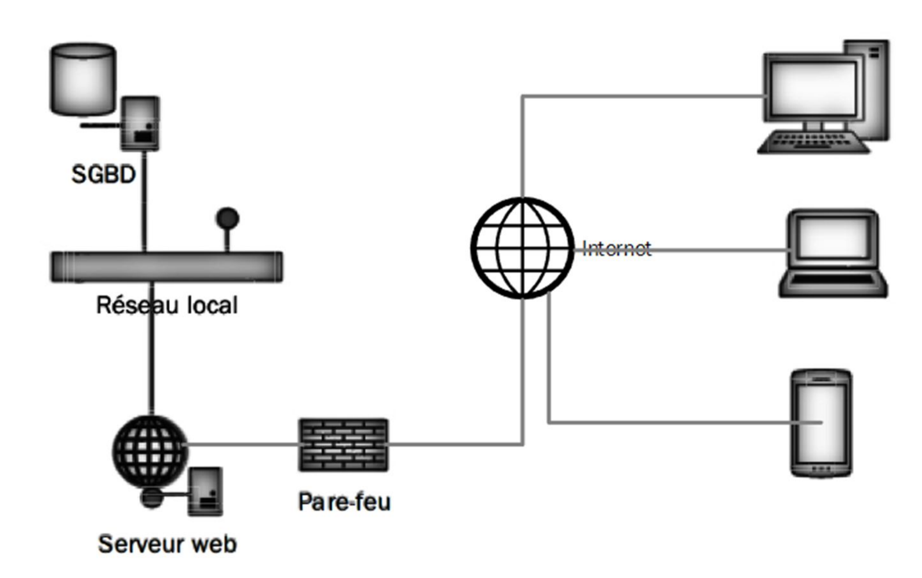


Figure 5 : architecture 3 niveaux

c) Utilisation d'un *proxy*

Lorsqu'une application doit consommer un service web, il est très pratique d'utiliser un *proxy* (1), une classe qui fournit les opérations du service web, au lieu de les demander directement. Ce modèle offre plusieurs avantages :

- Il est facile de tester l'application en utilisant un « faux » service web, ce qui évite les problèmes réseaux dans les tests
- Le reste de l'application est indépendant du service web, et utilise simplement un objet « normal »
- Si plusieurs applications utilisent le même service web, la classe est déjà écrite
- Le *proxy* peut mettre les données en mémoire (cache) et donc limiter les échanges réseaux (lents et pouvant être coûteux)

Le modèle est le suivant :

- Une interface représente les opérations que le service web peut effectuer
- Une classe implémente cette interface et effectue les requêtes *http* et la mise en forme des réponses
- Une autre classe (le *proxy*) implémente également cette interface et utilise la classe précédente pour les appels réseau. Elle peut ajouter un cache pour accélérer les traitements. Il arrive que les deux classes soient confondues en une.

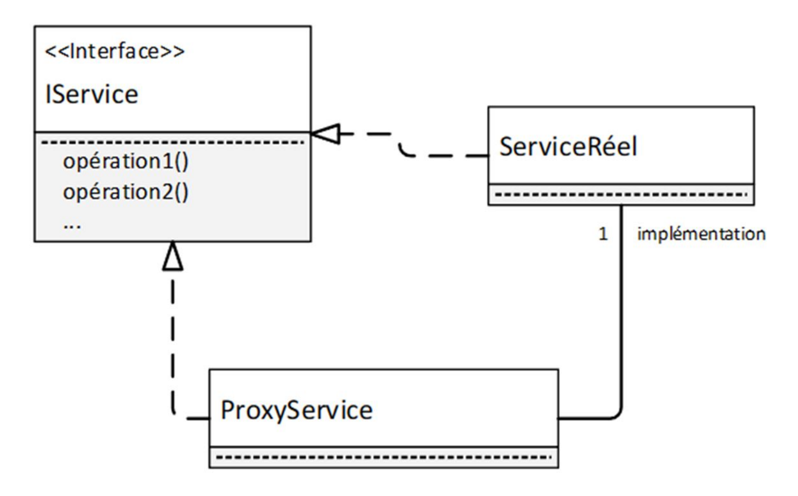


Figure 6 : modèle du proxy pour service web

d) Analyse de la réponse du service web

Le service web est interrogé par une requête *http* et répond par un simple texte. Nous aurons donc une chaîne de caractères qu'il faudra analyser. Cette chaîne est souvent dans un format de données standard (XML, JSON…) mais ce n'est pas obligatoire et il faut se reporter à la documentation du service web.

Nous avons deux possibilités pour cette analyse :

- L'analyse directe de la chaîne par code : obligatoire si le protocole est non-standard, possible avec des formats comme HTML, XML ou JSON. Peut demander une quantité importante de code suivant le format.

- La désérialisation[1] de l'objet fourni : possible avec les formats standard XML et JSON, on obtient un objet contenant les données fournies. Le langage et/ou le *framework* utilisé doit rendre cette désérialisation possible.

3. Exemple de service web

Nous allons ici décrire un service web volontairement très simple.

Notre service réalisera l'addition de deux nombres.

Nous allons utiliser le langage PHP (d'autres technologies seraient bien entendu possibles) : notre service pourrait donc être hébergé par n'importe quel serveur web utilisant un interpréteur PHP. Vous pouvez le tester sur votre machine en utilisant un outil comme XAMP.

Notre protocole sera très simple :

- Le service ne supportera que les requêtes *GET*
- La fonction s'appelle `add` et les paramètres `a` et `b`
- Le retour s'effectue en XML. La variable retournée s'appelle `sum`. Elle sera la valeur du nœud XML.

Le service peut donc être représenté par la classe ci-contre :

ServiceWeb
+Add(a : number, b:number) : number

Le code PHP du service est le suivant :

```
<?php
if(isset($_GET["a"])&&isset($_GET["b"])&&isset($_GET["func"])
      )
{
      $f = $_GET["func"];
      if($f=="add")
      {
        $a = $_GET["a"];
        $b = $_GET["b"];
        $sum = $a+$b;
```

[1] Sérialiser un objet revient à transformer un objet en une suite de caractères contenant les mêmes données (souvent une chaîne, donc) dans le but de le stocker ou de le transmettre via un réseau. La désérialisation est l'opération inverse, transformer une suite de caractères en un objet mémoire.

```
        echo '<?xml version="1.0" encoding="UTF-8"?>';
        echo "<sum>$sum</sum>";
    }
    else{
        echo 'Seule la fonction add est supportée';
    }
}
else
{
    echo 'Service web simple :
    service.php?func=add&a=1&b=2';
}
?>
```

La consommation de ce service se fait tout simplement par une requête *GET* comme par exemple pour additionner `10` et `3` (la requête peut être faite dans un navigateur) :

```
service.php?func=add&a=10&b=3
```

Le texte du retour du service est le suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<sum>13</sum>
```

4. Exemple de service web existant

Le site `hostpi.info` fournit des services web simples et gratuits pour localiser une adresse IP (principalement aux USA). Pour obtenir, par exemple, la localisation de l'adresse IP 8.8.8.8 (google) en JSON :

```
http://api.hostip.info/get_json.php?ip=8.8.8.8
```

Le retour à cette requête est le suivant :

```
{"country_name":"UNITED
    STATES","country_code":"US","city":"Mountain View,
    CA","ip":"8.8.8.8"}
```

Vous pouvez utiliser ce service pour tester l'appel *http* et l'analyse JSON décrits dans le chapitre ci-dessous.

5. Programmer les requêtes *http*

Dans une application qui va consommer un service web, nous allons par conséquent devoir exécuter une requête *http* et analyser sa réponse. L'outil utilisé pour développer l'application doit donc nous fournir des opérations permettant de jouer le rôle d'un client *http*.

a) Programmation asynchrone

Une requête *http* peut être longue à traiter, pour diverses raisons : lenteur du réseau utilisé, occupation ou surcharge du serveur, etc. Il est donc nécessaire de faire des appels **asynchrones**, c'est-à-dire non bloquants. En effet, si notre requête *http* était bloquante pour l'application, celle-ci ne répondrait plus à l'utilisateur (pas de mise à jour de l'affichage, par exemple) qui pourrait croire qu'elle a « planté » et qu'il faut la redémarrer. Les outils utilisés pour faire des appels réseau (donc des requêtes *http*) sont tous asynchrones.

La programmation « classique » est synchrone : quand on appelle une fonction, on attend son résultat pour continuer, comme dans le pseudo-code ci-dessous :

```
Fonction1()
// le code suivant est exécuté une fois Fonction1 terminée
```

En programmation asynchrone, c'est différent : l'appel d'une fonction se déroule en parallèle, et le programme continue. Il faut donc mettre en place des techniques pour réagir à la fin de cette fonction. Suivant le langage et l'outil, les techniques peuvent être :

- Utilisation de fonctions de rappel, transmises à la fonction asynchrone (via pointeur de fonction, foncteur, interface particulière…)
- Modèle observateur (1)
- Intégration directe dans le langage des fonctions asynchrones

b) Consommation d'un service web avec Java/Android

Android interdit d'utiliser des appels réseau dans le *thread* principal, consacré à l'affichage de l'IHM. Il faut donc pouvoir exécuter les différentes

requêtes dans un *thread* différent. Notre classe effectuant les requêtes doit donc implémenter l'interface `Runnable` (il est possible également de la faire hériter de `Thread` mais cela est moins pratique, et c'est, de toute façon, impossible si notre classe possède déjà un ancêtre).

Les API Android fournissent la classe `HttpURLConnection` qui permet de se connecter à un serveur *http* et d'effectuer une requête GET (par l'URL donc). Pour récupérer une instance de cette classe, il faut passer par un objet de type URL auquel on transmet une chaîne contenant l'URL :

```
URL u = new URL(url);
HttpURLConnection connection =
     (HttpURLConnection) u.openConnection();
connection.connect();
```

Pour récupérer la réponse à notre requête, la classe `HttpURLConnection` nous fournit un flux (`InputStream`) ; en utilisant les fonctions standard de Java de traitement des flux et des chaînes, nous pouvons aisément récupérer la chaîne complète de la réponse :

```
String reponse = "";
BufferedReader reader = null;
try {
     StringBuffer buffer = new StringBuffer();
     InputStream is = connection.getInputStream();
     reader = new BufferedReader(new InputStreamReader(is));
     String line;
     while ((line = reader.readLine()) != null) {
       buffer.append(line);
     }
     reponse = buffer.toString();
} finally {
     if (reader != null) reader.close();
     connection.disconnect();
}
```

Cette chaîne peut ensuite être analysée, suivant son format, pour récupérer les données voulues. Par exemple, nous pouvons utiliser les classes `JSONObject` et `JSONArray` pour désérialiser directement du JSON. Nous allons le voir dans un exemple. Soit le texte JSON suivant :

```
{"Nom" :"Tigrou", "Espece" :"Chat", "Race" :"Européen" }
```

Ce qui correspond à la classe Java :

```
class Animal {
     private String nom ;
     private String espece ;
     private String race ;
     public String getNom(){return nom ;}
     public void setNom(String value){nom=value ;}
     public String getEspece(){return espece ;}
     public void setEspece(String value){espece=value ;}
     public String getRace(){return race ;}
     public void setRace(String value){race=value ;}
}
```

Le code suivant permet d'initialiser l'objet à partir de la chaîne `str` contenant le JSON :

```
JSONObject obj = new JSONObject(str) ;
Animal a = new Animal() ;
a.setNom(obj.getString("Nom")) ;
a.setEspece(obj.getString("Espece")) ;
a.setRace(obj.getString("Race")) ;
```

c) Consommation d'un service web avec Swift / iOS

Swift utilise quelques classes pour faire l'appel *http* :

- `URLRequest` pour la requête *http* elle-même
- `URLSession` pour la connexion *http*
- `Task` pour la gestion asynchrone de la connexion

Dans le code suivant, la requête *http* stockée dans la chaîne `str` est effectuée, les données de la réponse sont stockées dans la variable `data` :

```
let url:URL = URL(str)!
let request:URLRequest = URLRequest(url : url)
let session = URLSession.shared
let task = session.dataTask(with : request)
{
     (data,response,error) in
     if (data != nil)
     {
        // traitement des données
     }
}
task.resume()
```

Pour des raisons de sécurité, iOS empêche la connexion *http* avec un serveur, sauf autorisation explicite dans le manifeste (`info.plist`). Vous pouvez spécifier explicitement le(s) domaine(s) autorisé(s), ou autoriser toute connexion (plus simple, mais moins sécurisé) comme sur la capture ci-dessous :



Une fois les données, récupérées, si celles-ci sont codées en JSON, il suffira d'utiliser la classe `JSONSerialization` pour récupérer un objet *swift* à partir des données.

d) Consommation d'un service web avec C# / .NET

Que ce soit pour Windows UWP, ou Android et iOS (avec Xamarin), le *framework* .NET contient les classes `HttpClient` et `HttpResponseMessage` qui permettent de se connecter à un service web et d'obtenir la réponse sous forme d'une chaîne de caractères. La sérialisation .NET permet en outre facilement de récupérer les données issues de XML ou JSON.

Les opérations du *framework* sont asynchrones, mais l'asynchronisme est géré nativement par C# grâce aux mots-clés `async` et `await`.

Le code suivant permet de récupérer une chaîne de caractères à partir d'une URL (requête *http GET*). Il doit être placé dans une fonction `async` :

```
HttpClient client = new HttpClient();
HttpResponseMessage reponse = await client.GetAsync(url);
String s = await reponse.Content.ReadAsStringAsync();
```

Pour récupérer un objet à partir du code JSON suivant :

```
{"Nom" :"Tigrou", "Espece" :"Chat", "Race" :"Européen" }
```

Il faut commencer par créer une classe C# sérialisable :

```
[DataContract] class Animal {
     [DataMember] public String Nom {get ;set ;}
     [DataMember] public String Espece {get ;set ;}
     [DataMember] public String Race {get ;set ;}
}
```

Et enfin, pour désérialiser le texte JSON contenant dans la chaîne `str` :

```
using(MemoryStream flux = new
    MemoryStream(Encoding.UTF8.GetBytes(str)))
{
    DataContractJsonSerializer ser = new
    DataContractJsonSerializer(typeof(Animal)) ;
    Animal a = ser.ReadObject(flux) as Animal ;
}
```

e) Consommation d'un service web avec Javascript

Un moyen simple d'exécuter une requête *http* en Javascript est d'utiliser JQuery (23) et la fonction `get` fournie.

Celle-ci prend deux paramètres : l'URL de la requête et la fonction de rappel qui est appelée lorsque la réponse est arrivée.

L'exemple suivant récupère le texte de la réponse à la requête contenue dans la variable `url` pour la placer dans la chaîne `result` et la traiter dans la fonction de rappel passée en paramètre (la fonction est ici définie en ligne) :

```
$.get(url, function(result){
//… traitement de la réponse : result
}) ;
```

Soit le texte JSON suivant :

```
{"Nom" :"Tigrou", "Espece" :"Chat", "Race" :"Européen" }
```

JavaScript peut évidemment désérialiser nativement du JSON contenu dans une variable `str` :

```
var a = JSON.parse(str) ;
```

f) Consommation d'un service web avec C++/Qt

L'asynchronisme avec Qt passe par le système de signaux/*slots*. Pour pouvoir s'en servir, une classe doit hériter de `QObject`. Elle va déclarer un *slot* pour représenter la fin de la requête :

```
Q_OBJECT
private slots:
    void requeteOk(QNetworkReply* result) ;
```

Les classes `QNetworkManager`, `QUrl` et `QNetworkRequest` sont fournies par Qt pour exécuter une requête *http*.

L'exemple suivant exécute la requête contenue dans la chaîne `url` :

```
QNetworkAccessManager manager;
QUrl u(url);
QNetworkRequest req(u);
connect( & manager, SIGNAL(finished(QNetworkReply * )), this,
     SLOT(requeteOk(QNetworkReply * )));
manager.get(req);
```

Dans le *slot,* il est aisé de récupérer dans une chaîne le résultat à partir du paramètre `result` :

```
result->deleteLater() ;
QString s = result->readAll() ;
```

Pour récupérer des données en JSON, il suffit d'utiliser les classes `QJsonObject` et `QJsonDocument`.

Soit le texte JSON suivant :

```
{"Nom" :"Tigrou", "Espece" :"Chat", "Race" :"Européen" }
```

Ce qui correspond à la classe C++ :

```
class Animal {
private:  std::string nom ;
       std::string espece ;
       std::string race ;
public:  std::string getNom() const {return nom ;}
        void setNom(std::string value){nom=value ;}
        std::string getEspece() const{return espece ;}
        void setEspece(std::string value){espece=value ;}
        std::string getRace() const{return race ;}
        void setRace(std::string value){race=value ;}
} ;
```

Le code suivant permet d'initialiser l'objet à partir de la chaîne `str` contenant le JSON :

```
QJsonDocument json = QJsonDocument::fromJson(str.toUtf8());
QJsonObject jsonObject = json.object(); Animal a ;
a.setNom(jsonObject["Nom"].toString().toStdString()) ;
a.setEspece(jsonObject["Espece"].toString().toStdString()) ;
a.setRace(jsonObject["Race"].toString().toStdString()) ;
```

XI. Déploiement d'une application

Déployer une application signifie la transférer sur l'appareil ciblé afin qu'elle soit opérationnelle pour l'utilisateur. Sur un téléphone, cela revient en général à mettre l'application sous une certaine forme, imposée par l'éditeur du magasin d'applications, afin qu'elle soit à la disposition des utilisateurs sur celui-ci.

Nous verrons dans ce chapitre la notion de **manifeste**, de *packaging* d'une application et comment déposer celle-ci sur un *store*.

1. Manifestes et ressources visuelles

Le **manifeste** d'une application est un fichier contenant un certain nombre d'informations sur l'application, informations qui sont nécessaires au système d'exploitation mais également au magasin d'applications (le *store*).

Les trois systèmes utilisent cette notion de manifeste, mais avec des légères variantes.

a) Android

Android utilise un fichier XML (`AndroidManifest.xml`) pour stocker son manifeste. Ce manifeste est organisé de la manière suivante :

- Le nœud XML principal est « `manifest` » et contient comme informations (attributs) le nom du *package* de l'application ainsi que son numéro de version
- Le sous-nœud « `application` » contient les composants de l'application : activités (`activity`), services, récepteurs (`receiver`) et fournisseurs (`provider`). Chaque classe de l'application héritant de `Activity`, `Service`, `BroadcastReceiver` ou `ContentProvider` doit avoir un sous-nœud dans « application ».
- Les nœuds « `icon` » servent à indiquer l'icône utilisée pour leur parent (application ou activité)
- Les nœuds « `uses-permission` » servent à indiquer les permissions nécessaires à l'application

La plupart des EDI remplissent automatiquement le manifeste ou proposent un éditeur visuel pour le faire.

b) iOS

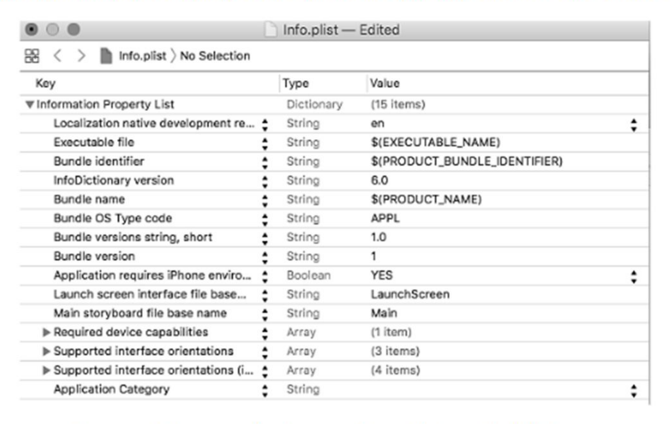
iOS utilise également un fichier de type XML pour son manifeste. Ce fichier s'appelle `infos.plist`. Plutôt que de l'éditer comme fichier XML, il est plus efficace de l'éditer avec XCode directement : l'édition est visuelle et plus facile.

Ce fichier contient au minimum les informations suivantes :

- La langue par défaut de l'application (`en` : anglais par défaut)
- Le nom du fichier exécutable
- L'identifiant du paquet (de la forme `com.entreprise.appli`)
- Le nom du paquet
- Le type de paquet (APPL pour une application)
- Le numéro de version du paquet (1.0 par exemple)
- Les capacités obligatoires du périphérique

C'est dans ce fichier que l'on trouvera également des informations liées au respect de la vie privée, notamment le texte à afficher à l'utilisateur pour indiquer les autorisations de l'application (localisation, photo, etc…).

XCode permet d'éditer ce manifeste, mais il est également modifiable avec les autres environnements, ou directement avec un éditeur de texte.



Key	Type	Value
▼ Information Property List	Dictionary	(15 items)
Localization native development re...	String	en
Executable file	String	$(EXECUTABLE_NAME)
Bundle identifier	String	$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle version	String	1
Application requires iPhone enviro...	Boolean	YES
Launch screen interface file base...	String	LaunchScreen
Main storyboard file base name	String	Main
▶ Required device capabilities	Array	(1 item)
▶ Supported interface orientations	Array	(3 items)
▶ Supported interface orientations (i...	Array	(4 items)
Application Category	String	

Capture 71: exemple de manifeste (infos.plist) iOS

c) Windows 10

Windows (du moins dans la plateforme universelle) demande également un manifeste pour les applications. Ce manifeste est lui aussi un fichier au format XML dont le nom est `Package.appxmanifest`. Ce fichier peut être édité avec un simple éditeur de texte mais il est beaucoup plus simple et efficace d'utiliser l'éditeur intégré à Visual Studio.

Le manifeste Windows comprend plusieurs sections :

Application Actifs visuels Capacités Déclarations URI de contenu Packages

Dans la première (`Application`) sont configurés :

- Le nom de l'application
- La classe principale (point d'entrée) de l'application
- La langue par défaut (`fr-FR` pour le français) de l'application
- Un texte de description
- L'indication (facultative) en cas de notifications de l'écran de verrouillage
- Les réglages (facultatifs) des vignettes de l'application (*tiles*)

La deuxième section (`Actifs visuels`) permet de régler les différentes images nécessaires à l'application. Une application UWP requiert de nombreuses images dont les tailles sont très précises. Un générateur est fourni, qui permet de créer toutes ces images à partir d'une seule. Le format doit être PNG (pour permettre la gestion de la transparence) et il est conseillé que l'image de départ ait une résolution suffisante (au moins 512x512).

La troisième section (`Capacités`) permet de choisir les fonctionnalités système que l'application peut utiliser. C'est ici que les autorisations nécessaires de l'application sont indiquées (par exemple : client Internet, Webcam, Emplacement…).

Les 4e et 5e sections sont plus rarement utiles. La 6e et dernière section est en revanche fondamentale pour la publication sur le *Microsoft Store*, mais elle est fréquemment remplie automatiquement.

2. Le *packaging*

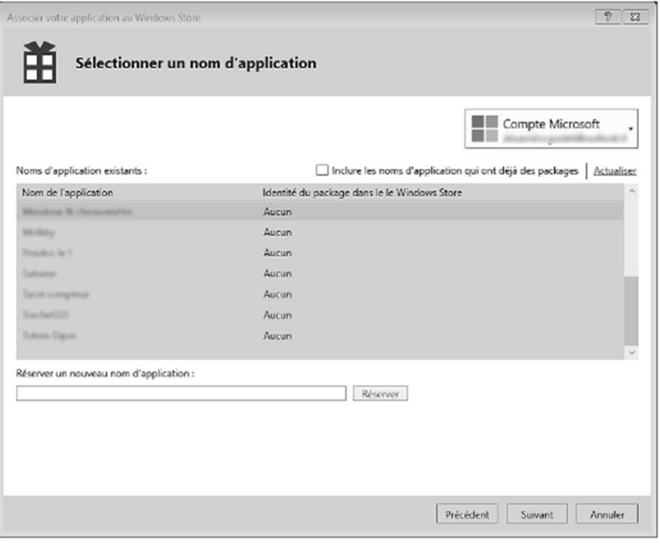
Les trois magasins d'applications (Microsoft Store, Apple Store, Google Play Store) demandent tous un format particulier pour l'application qui doit être fournie sous forme d'un seul fichier empaquetant l'ensemble de l'application. Le format de ce fichier varie suivant le magasin, mais dans tous les cas, il répond à certains points :

- Il contient :
  - Tous les fichiers nécessaires à l'application, pour toutes les architectures ciblées
  - Un numéro de version obligatoirement supérieur au précédent pour faciliter la mise à jour automatique
  - Un identifiant de paquet pour identifier de manière unique sur le magasin l'application
- Il est signé par un certificat pour garantir que l'auteur d'une mise à jour est le même que l'auteur initial

Pour publier l'application, il est donc nécessaire de l'empaqueter dans le bon format suivant le protocole demandé par le *store*. Ceci dépend à la fois de l'outil utilisé et du magasin d'applications.

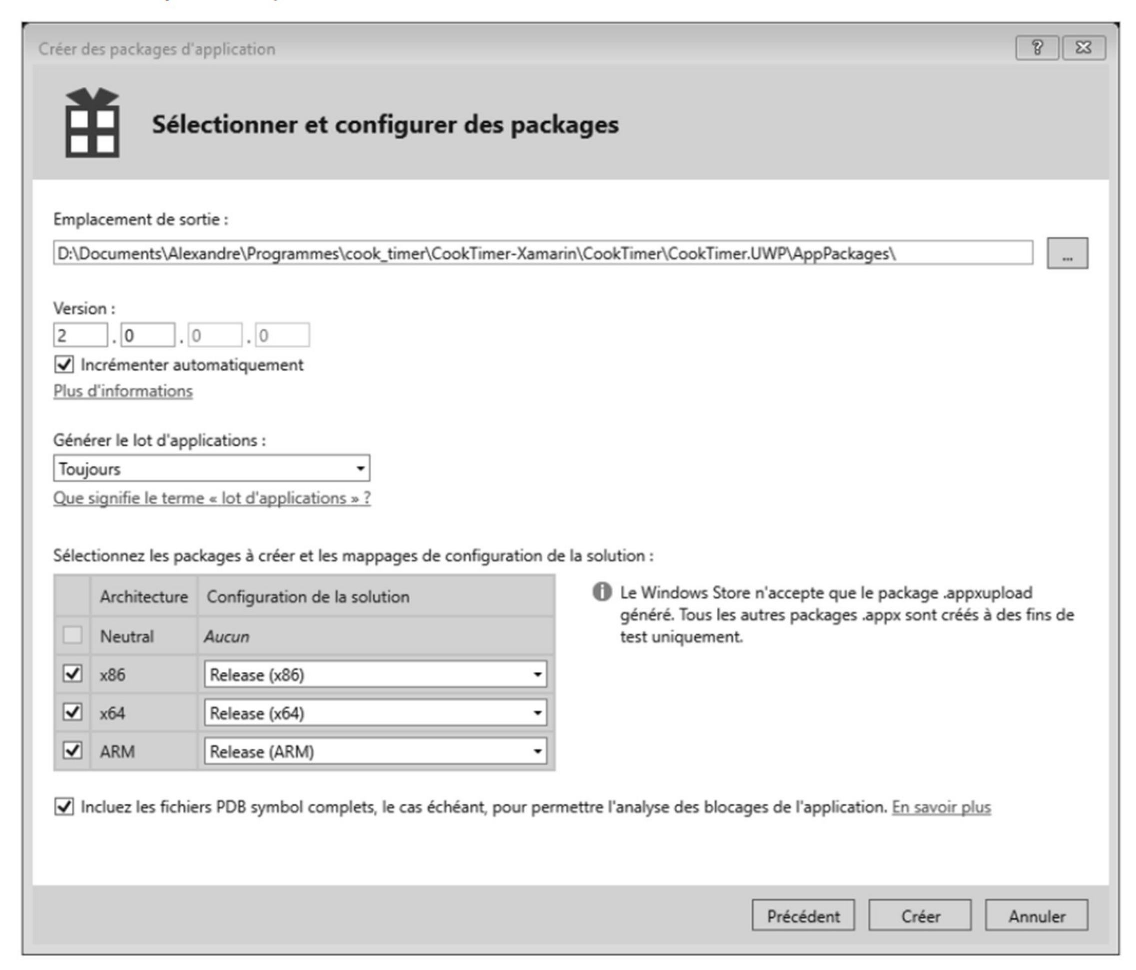
a) Visual Studio - *Packaging* pour Windows

Pour créer un *package* à déposer sur le Windows Store, il faut déjà (en plus d'avoir un compte développeur Microsoft) créer une application sur le site développeur Windows. Une fois l'application créée, il est très simple de l'associer au projet : il suffit de cliquer (droit) sur le projet UWP, de choisir « Store → associer l'application au store » et ensuite choisir le nom de l'application associée (si vous n'avez pas encore créé d'application sur le site, vous pouvez le faire ici).



Capture 72 : associer une application UWP à votre projet

Une fois l'application associée, il convient de refaire un clic-droit sur le projet et de choisir « Store → Créer des packages d'application ». Vous pouvez ensuite choisir le dossier de sortie des *packages* (notez-le bien), le numéro de version (choisissez-le bien) ainsi que les architectures choisies (x86 pour les PC/tablettes 32 bits, x64 pour les PC/tablettes 64 bits, ARM pour les téléphones) :



Capture 73 : génération des packages d'une application UWP

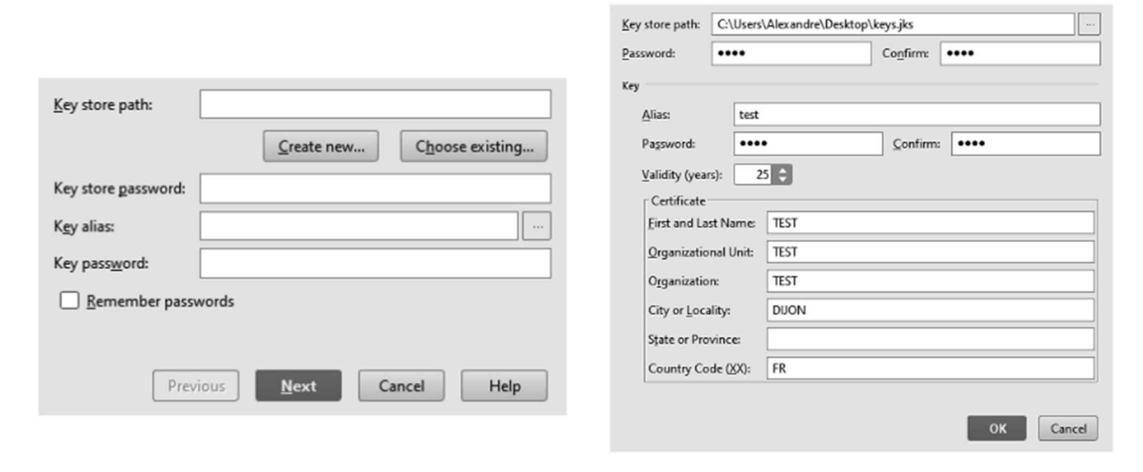
Patientez : la création du *package* prend un peu de temps. Une fois celle-ci finie, (si elle a réussi) Visual Studio vous propose de réaliser un certain nombre de tests : acceptez ! Si les tests échouent, votre application ne sera pas acceptée sur le *store*…

Ne touchez pas votre ordinateur pendant les tests : l'application sera lancée, fermée, relancée… Une fenêtre vous avertira de la réussite ou de l'échec des tests.

Une fois tous les tests passés, un fichier `.appxupload` est créé dans le dossier précisé : ce fichier sera utilisé pour déposer l'application sur le *Windows Store* (XI.3.b).

b) Android Studio – Packaging pour Android

Une fois l'application finalisée, y compris le manifeste, les ressources visuelles, etc. il suffit de créer un fichier APK de distribution. Pour cela, dans Android Studio, il faut choisir le menu *Build* → *Generate Signed APK*. Il faut ensuite spécifier le chemin du fichier de clés pour signer le paquet, ou en créer un nouveau (bien le conserver, il sera nécessaire pour créer une mise à jour du programme).

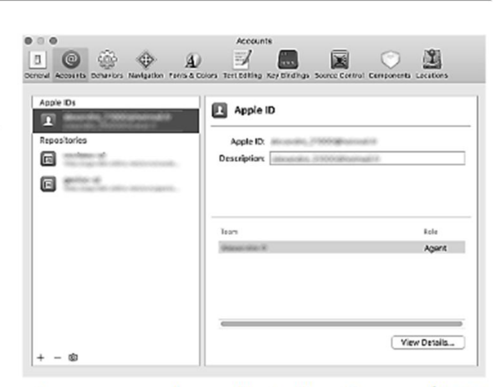


Capture 74 : signature paquet Android

Il reste à spécifier le dossier de sortie de l'APK généré, en choisissant le type *release*. Une fois le fichier généré, il peut être déposé sur le Google Play Store.

c) XCode – *Packaging* pour iOS

Une fois l'application finalisée, et testée en mode *release* (y compris sur un appareil réel) il est possible de la *packager* pour une publication sur l'*Apple store*.

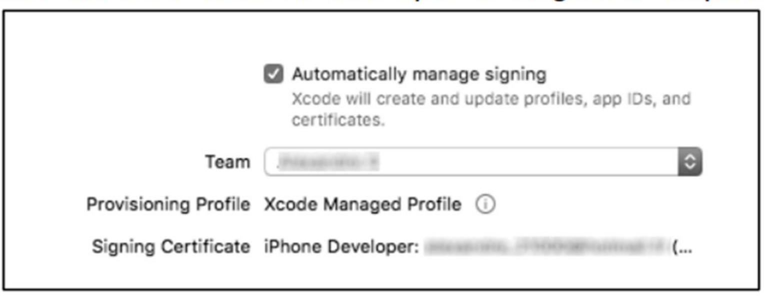


XCode doit bien entendu être relié à votre compte développeur Apple. Pour cela, il faut aller dans le menu *XCode* → *Preferences*, onglet *Accounts*. Cliquez sur « *view details* » pour afficher les détails de votre compte. Vous devez avoir des certificats (*signing identities*) pour le développement (*iOS Development*) mais aussi pour la distribution (*iOS Distribution*). S'il en manque, faites-les générer en cliquant sur « *create* ».

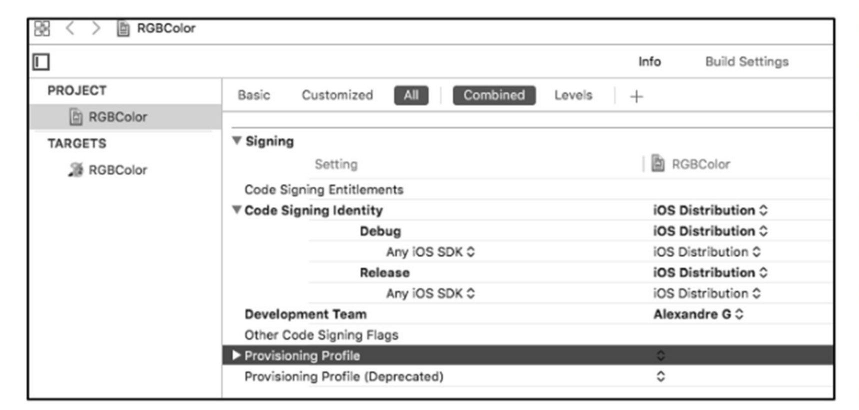


Sur le site développeur d'Apple, il faut ensuite créer un *AppID* pour votre application. Choisir un nom clair, et un identificateur de lot (*bundle id*) qui correspond à celui entré dans *XCode* (par exemple, `com.ag.rgbcolors`). Validez.

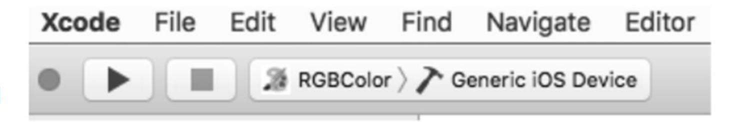
Il faut ensuite créer un **profil d'approvisionnement** pour votre application. Allez à la rubrique « *Provisionning profiles* » du site *Apple Developer* ou laissez faire XCode : dans la rubrique « *Targets* » cliquez « *automatically manage signing* » pour que l'application soit signée avec votre certificat. Testez en déployant l'application sur un appareil (iPhone ou iPad).



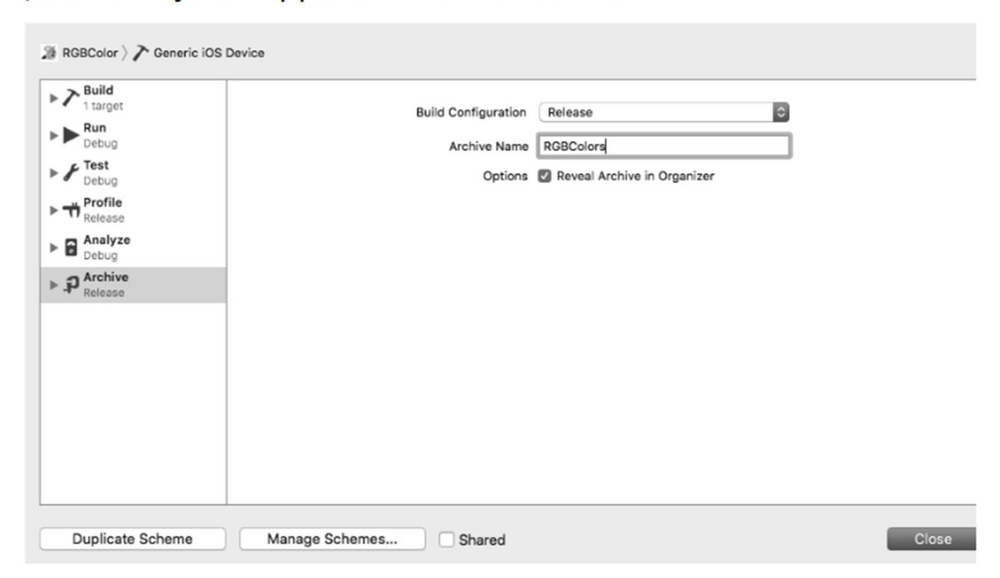
Ouvrez ensuite les propriétés de votre projet XCode, choisir *Build Settings, All* et la rubrique *Signing*. Choisissez votre profil développeur, et configurez sur *iOS Distribution*.



Faites ensuite un nettoyage du projet (*menu product →clean*). Choisir ensuite *Generic iOS device* dans la liste déroulante des cibles (barre du haut).



Choisir ensuite le menu « *Product →scheme →edit scheme* ». Entrez dans l'onglet « *archive* », vérifiez le nom de votre application et que la case « *reveal archive* » soit cochée. Lancez ensuite la commande « *product →archive* ». Une fois l'archivage terminé (si tout va bien) votre archive est visible dans l'organiseur d'XCode (*window →organizer*). Le bouton « *upload to app store* » permet de publier/mettre à jour l'application directement.



Le *package* est à présent prêt à être déposé sur le *store*.

d) Qt – *packaging* pour Windows

Qt ne peut pas créer directement un paquet d’installation pour Windows universel. Il ne fait que créer un dossier qui contient tout ce qu’il faut pour construire le *package*.

La première chose à faire est de trouver ce dossier (il est indiqué dans les paramètres du projet) et d’éditer le manifeste (`AppxManifest.xml`) pour renseigner les valeurs demandées par le *Microsoft Store* (ID de l’éditeur, de l’application).

Les outils du SDK Windows 10 (qui doit donc être installé) sont ensuite utilisés. Si votre application Qt existe pour plusieurs plateformes (x86, x64, arm, arm64…), il faudra faire ces étapes <u>pour chacune d’entre elles</u>.

Il faut ensuite utiliser l’application `MakeAppx` du SDK Windows 10[1] en lui donnant en paramètre le chemin du dossier créé par Qt Creator et le nom du fichier *package* souhaité :

```
MakeAppx pack /d <dossier> /p <package>
```

Le paquet généré n’est pas encore signé et ne peut donc pas être diffusé sur le Windows Store.

Pour votre premier dépôt, il faut créer un certificat auto-signé correspondant à l’identité de l’éditeur voulue, dans une console PowerShell[2] :

```
New-SelfSignedCertificate -Type Custom -Subject <ID de l’éditeur>
     -KeyUsage DigitalSignature -FriendlyName <Nom de l’éditeur> -
     CertStoreLocation "Cert:\LocalMachine\My"
```

La commande donne en réponse une empreinte (*Thumbprint*) qu’il faut <u>conserver pour la suite</u> (à copier/coller dans un bloc-notes par exemple).

---

[1] Le SDK Windows 10 est normalement installé avec Visual Studio. Son chemin standard est `c:\program files (x86)\Windows Kits\10\bin\<version du SDK>\x64`.

[2] Sous Windows 10, la ligne de commande est remplacée par PowerShell qui est un interpréteur avancé de commandes. Pour le lancer, il suffit de taper Powershell dans la barre de recherche.

Le certificat généré est installé dans la machine locale. Il faut l'exporter pour pouvoir l'utiliser pour signer l'application. Pour cela, il faut choisir un mot de passe et utiliser les commandes *Powershell* suivantes :

```
$pwd = ConvertTo-SecureString -String <nom> -Force -AsPlainText
Export-PfxCertificate -cert Cert:\LocalMachine\My\<thumbprint> -
    FilePath <nom fichier>.pfx -Password $pwd
```

Vous obtenez alors un fichier à utiliser avec l'outil `SignTool` du SDK Windows 10 :

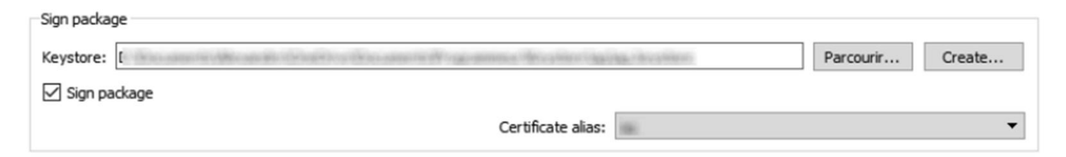
```
signtool.exe sign /fd SHA256 /a /f <nom du fichier certificat>.pfx
    /p <mot de passe du certificat> <chemin du fichier appx>
```

Le fichier est à présent signé et peut être déployé.

Le certificat peut être employé pour signer plusieurs applications : conservez-le et n'oubliez pas son mot de passe !

e) Qt – *packaging* pour Android

Qt Creator possède toutes les options pour créer et signer le *package* Android directement (il ne faut pas oublier de créer le manifeste). La compilation produira le fichier APK qui est destiné à être publié sur le *Play Store*. Il ne faut pas oublier de signer le package avec le *keystore* fourni sur le *Google Play Store*.



Attention si vous faites une mise à jour : le certificat employé doit être le même que pour la première version ! Stockez donc bien le fichier *keystore* et n'oubliez pas son mot de passe !

f) Qt – *packaging* pour iOS

Qt ne possède pas la possibilité (du moins pour l'instant, dans sa version gratuite) de créer directement un *package* pour l'Apple Store. En revanche, il génère un projet pour XCode : le *packaging* peut donc être réalisé directement avec XCode, comme vu page 250. Il est en revanche nécessaire

d'éditer le projet XCode et le manifeste car Qt ne les remplit par complètement (notamment les images).

g) Xamarin – *packaging* pour Windows

Visual Studio étant l'environnement natif pour le développement Windows, le *packaging* de l'application pour Windows est très simple : voir page 248.

h) Xamarin – *packaging* pour Android

Une fois l'application prête et le manifeste correctement renseigné, il ne reste plus qu'à construire le fichier APK destiné à être envoyé au Google Play Store.

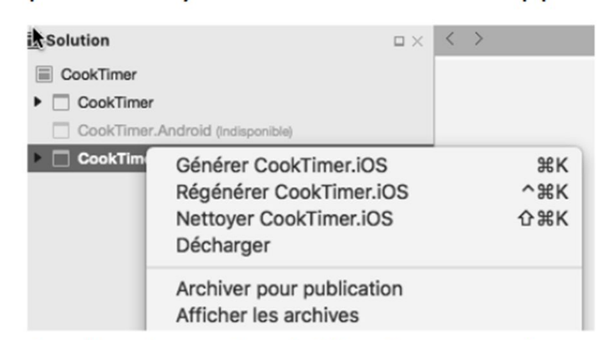
Visual Studio contient tout le nécessaire pour cela, il suffit de faire un clic-droit sur le projet Android et de sélectionner « Archiver » (attention : bien se placer en mode *release* et non *debug*).

Une fois l'archive construite, il reste à la signer pour la placer sur le Google Play Store. Il faut donc cliquer sur le bouton [Distribuer...] et choisir le canal « *Ad Hoc* ». Il faut choisir ensuite une identité de signature. Si vous en possédez déjà une, autant la choisir, sinon il faut la créer (bouton +) et bien conserver le fichier obtenu ainsi que le mot de passe choisi, ils seront utiles pour la mise à jour de l'application. Cliquez enfin sur « enregistrer sous » pour choisir le dossier où l'APK sera placé.

i) Xamarin – *packaging* pour iOS

Le projet doit être signé pour iOS, comme décrit page 60 (notamment la création de l'AppID et du profil d'approvisionnement). Réglez la configuration du projet sur « *AppStore* ». Réglez bien le manifeste (`infos.plist`) et la liste des autorisations (`Entitlements.plist`). Dans les options de construction, choisissez l'architecture ARM64. Cochez la case « options IPA → générer une archive IPA » et décochez la case « inclure les images iTunes ».

Faites régénérer la solution. Votre fichier IPA[1] est prêt, sur la machine ayant servi à compiler (le Mac donc). Sur le Mac, utilisez *Application Loader* pour envoyer le fichier IPA sur l'Apple Store.



Une autre solution est d'ouvrir la solution directement sur le mac avec Visual Studio et de générer l'archive (clic-droit sur le projet → archiver pour publication), le déploiement pourra se faire ensuite directement depuis Visual Studio. Attention à bien importer le profil d'approvisionnement créé depuis le site *Apple Developer,* ou utiliser un provisionnement automatique. Voir (24) et (25) pour plus de détails.

j) Cordova – packaging pour Windows

Visual Studio étant l'EDI « normal » pour le développement Windows, la gestion du *packaging* et de la publication en est grandement facilitée !

La création est identique à celle vue au XI.2.a).

k) Cordova – packaging pour Android

Il faut commencer par préparer l'application en éditant le fichier `config.xml` à la racine du projet. Dans l'onglet « `commun` », indiquez les informations *ad hoc*. Faites de même dans l'onglet `Android` en indiquant au minimum :

- Le code de version (il doit être différent à chaque nouvelle version)
- Le niveau d'API minimal (entier – indique l'API Android nécessaire au minimum pour exécuter l'application)
- Le niveau d'API cible

Il faut ensuite avoir un certificat privé. Si vous en avez déjà un vous pouvez vous en servir, sinon vous devez en créer un. Pour cela, utilisez l'outil `keytool` du JDK :

---

[1] IPA : format de fichier paquet pour iPhone/iPad, archive contenant les exécutables et des fichiers annexes (manifeste, images…).

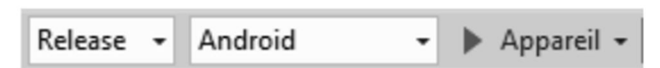
```
keytool -genkeypair -v -keystore monfichier.keystore -alias
     monalias -keyalg RSA -keysize 2048 -validity 10000
```

L'outil demandera un certain nombre d'informations à compléter (nom, pays…) ainsi qu'un mot de passe : bien le choisir et ne pas l'oublier !

Editez ensuite le fichier `build.json` ; à la section `android`, indiquez le chemin du fichier `keystore`, l'alias choisi et le mot de passe.

```
"android": {                                              {JSON}
    "release": {
      "keystore": "c:\\temp\\test.keystore",
      "storePassword": "XXX",
      "alias": "test",
      "password": "XXX",
      "keystoreType": ""
    }
```

Pour créer le paquet, il ne reste plus qu'à régénérer la solution en prenant la plateforme Android, configuration *release* ainsi qu'un périphérique réel comme cible :



Une fois la génération terminée (si réussie) vous trouverez dans le dossier `bin\android\release` deux fichiers `apk`. Celui ne contenant pas `unaligned` est le fichier à déployer sur le store.



I) Cordova – packaging pour iOS

Le plus simple pour réaliser le *packaging* est de le faire avec XCode. Il suffit donc de générer l'application en mode *release* avec Visual Studio, de récupérer le projet XCode généré et de le transférer sur un Mac.

Une fois ouvert avec XCode, si tout va bien, les démarches sont les mêmes que celles indiquées page 250.

3. Publier une application

Une fois l'application terminée, testée (sur simulateur et sur appareil physique), vous pouvez envisager une publication dans les magasins d'applications. L'étape de *packaging* étant indispensable avant de publier, lisez bien le chapitre précédent !

a) Pour Android : le Google Play Store

Avant de publier sur le Google Play Store, il faut s'inscrire auprès de Google comme développeur. Pour cela il faut un compte Google (si vous n'en avez pas il est simple et gratuit d'en créer un) et se rendre sur :

https://play.google.com/apps/publish/signup.

Une fois connecté, vous devez accepter le contrat du développeur et régler les frais d'inscription (25$) par carte bancaire. Ces frais ne sont dus qu'une seule fois, votre compte développeur reste actif à vie.

Pour publier une nouvelle application, il faut tout d'abord dans le tableau de bord (*google play console*) créer une application en cliquant sur le bouton *ad hoc* :

CRÉER UNE APPLICATION

Choisissez la langue par défaut ainsi que le titre de l'application.

Si le titre n'est pas déjà pris, la page suivante s'ouvre et vous demande une description (brève et complète) de votre application. Vous devez aussi fournir des captures d'écrans de l'application qui serviront à la visualiser dans le magasin. Une icône en haute résolution (512x512) ainsi qu'une large image (1024x500) sont également obligatoires.

Choisissez ensuite choisir le type d'application (application ou jeu) et sa catégorie. Vous devrez en outre indiquer une adresse e-mail, éventuellement un site web. N'oubliez pas d'enregistrer le brouillon pour confirmer vos saisies.

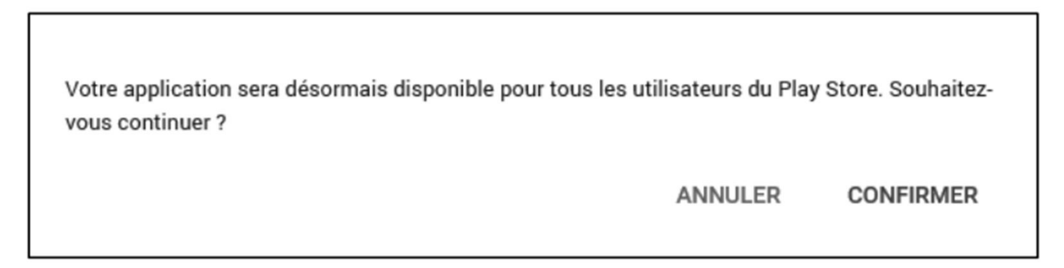
Déposez ensuite le fichier APK construit (voir le chapitre précédent, page 248) : choisissez « Versions de l'application », puis dans la partie « Production » (pour une application finie) ou « béta » (pour une application en test), cliquez sur « Gérer », puis « Créer une version », « Parcourir les fichiers » et allez chercher le fichier APK produit auparavant. Si tout va bien,

il ne reste plus qu'à valider. L'application n'est pas prête pour autant, il faut ensuite choisir « Classification du contenu », indiquer une adresse e-mail valide et répondre au questionnaire. Cliquez « Attribuer une classification », le site vous propose alors une classification que vous pouvez valider (vous recevrez un mail, à conserver).

Il ne reste plus qu'à choisir « Tarifs et disponibilité » pour finaliser. Si tout s'est bien passé, votre application est indiquée comme « Prête à être publiée » en haut du tableau de bord.



Retournez alors dans « Versions de l'application », puis « Modifier la version », « Vérifier » et enfin « Lancer le déploiement en version production ».



Et voilà ! Après quelques heures (au pire, quelques jours) l'application est visible dans le magasin !

b) Pour Windows : le Microsoft Store

Pour publier une application sur le magasin d'applications Microsoft (pour Windows 10), il est nécessaire de créer un compte développeur.

Il faut aller sur le site https://developer.microsoft.com/fr-fr/store/register et choisir « s'inscrire ». Utilisez votre compte Microsoft existant ou créez un nouveau compte Microsoft. Un compte personnel (pas d'entreprise) coûte 19$, à régler une seule fois et valable à vie.

Une fois votre compte développeur créé, vous avez accès à votre tableau de bord. Au début, il n'y a pas grand-chose, mais ici vous verrez vos applications, leurs statistiques, etc.

Commençons par créer une application.

Créer...
Créer une application
Créer un jeu

La première chose est de trouver un nom pour l'application. Ce nom doit être unique dans le Windows Store : lien « vérifier la disponibilité » permet de s'en assurer.

Créer votre propre application en réservant un nom

Une fois que vous réservez un nom, votre application sera mise en service pour des services comme les notifications push et vous pourrez commencer à définir des composants additionnels.

Assurez-vous de disposer des droits nécessaires pour utiliser un nom que vous réservez. Vous devez soumettre cette application au Store dans les trois mois, sinon vous perdez votre réservation de nom. En savoir plus

RGB Colors — Vérifier la disponibilité

Réserver le nom de produit — Annuler

Le nom de l'application est ensuite réservé. Il sera « bloqué » pendant 3 mois : si vous ne publiez pas votre application d'ici là, le nom sera libéré.

Une application est publiée en plusieurs soumissions : une soumission est en quelque sorte une version de l'application. Il faut donc démarrer une nouvelle soumission !

Démarrer la soumission

La soumission a besoin d'un certain nombre d'informations pour être transmise au magasin :

- **Tarif** et liste des pays dans lesquels l'application sera proposée
- Propriétés de l'application : **catégorie**, politique de confidentialité, etc.
- **Âge légal** pour autoriser l'application (nécessite de répondre à un questionnaire)
- ***Packages*** de l'application
- **Descriptions** (textes, images) pour le magasin

A chaque point, complétez les informations demandées. Quand tous les points seront validés, la soumission sera possible.

Le point le plus important est le transfert des *packages* d'application. Lisez bien la partie concernant votre outil dans le chapitre XI.2.

Une fois la publication lancée, si aucun problème ne survient, votre application est disponible sur le *store* en moins de 48h.

Soumettre au Store

c) Pour iOS : l'Apple Store

Normalement vous devez déjà avoir un compte développeur Apple (si ce n'est pas le cas, il est grand temps d'en créer un !). Connectez-vous sur le site *developer.apple.com/programs* et suivez les instructions pour créer votre compte. Il vous en coûtera **99€ par an**, avec renouvellement automatique : faites attention si vous ne voulez plus être développeur, vous pouvez annuler le renouvellement jusqu'à 24h avant la date anniversaire de votre inscription.

Apple est -de loin- le plus coûteux des 3 *stores*, aussi ne vous inscrivez à celui-ci que si vous êtes certain de vouloir déployer des applications pour iOS.

Au bout de quelques minutes, vous recevrez un *e-mail* vous informant que votre compte développeur est ouvert. Vous pouvez aller sur votre tableau de bord pour commencer à créer des applications iOS !

Il est à noter qu'Apple ne traduit pas toujours ses applications ou ses sites…

Pour gérer les applications, il faut aller sur « *app store connect* » (https://appstoreconnect.apple.com/) : ce site est, lui, traduit en français.

Pour publier une nouvelle application (ou une mise à jour), il faut aller sur « mes apps ». Cliquez sur le + pour créer une nouvelle application. Vous aurez le choix entre créer une nouvelle app (pour iPhone/iPad) ou une application (pour Mac).









Les identifiant de lots sont créés à partir des projets *XCode* reliés à votre compte. Si vous ne créez pas l'application à l'aide de XCode, il faut créer l'AppID sur le site développeur Apple et l'utiliser dans votre EDI (25).



Dans les informations de l'application, nous retrouvons son nom (déjà choisi), un éventuel sous-titre, la catégorie de l'application ainsi que divers informations pratiques.

La partie « Tarifs et disponibilité » permet d'indiquer le prix de l'application et les pays dans lesquels elle est délivrable.

La partie « Finaliser avec soumission » est celle où vous pouvez décrire votre application, placer des captures d'écrans, etc. Soignez bien cette partie ! Attention aux dimensions pour vos captures d'écran : pour un affichage 5.5 pouces (iPhone), Apple impose une résolution 1242x2208 pour une capture d'écran et de 1024x1024, sans transparence, pour l'icône pour le *store*.

Il ne faut pas oublier également de fournir une URL d'assistance (une page web donnant des conseils pour utiliser votre application).

Une fois que toutes les informations sont rentrées, il faut encore téléverser l'archive créée : retournez dans *XCode*, dans l'organiseur, onglet archives, et choisir « *upload to app store* ». Une fois ce téléversement terminé, retournez sur le site de *l'Apple Store*. Allez sur « Finaliser avant… », paragraphe *build* et choisir une *build* téléversée (sa validation peut prendre quelque temps) puis choisir « soumettre pour validation ». Il ne vous reste plus qu'à patienter : Apple va vérifier que votre application est conforme, ce qui peut prendre du temps (deux semaines maximum) et, si c'est le cas, la publier sur le *store* ; dans le cas contraire, vous recevrez un *e-mail* indiquant les causes de la non validation de votre soumission. Il faudra corriger celle-ci avant de recommencer. La validation peut nécessiter du temps, soyez patients !



XII. Applications

Nous arrivons dans la partie pratique de cet ouvrage ! Dans ce chapitre, nous allons développer 10 applications pour téléphones/tablettes. Chacune de ces applications est un exemple concret d'un chapitre de ce livre, parfois de plusieurs. J'ai choisi un outil pour chacune de ces applications, mais vous pouvez (c'est même un exercice intéressant) choisi un autre outil pour la développer !

Pour mieux visualiser chaque application, celles-ci sont déployées sur les magasins d'applications de chaque plateforme (Android, Windows, iOS). Le code source complet est également téléchargeable.

Les ateliers sont présentés par ordre de difficulté croissante, bien que ce ne soit qu'une simple indication, la difficulté étant très subjective.

Les applications sont volontairement limitées dans leurs fonctionnalités, leur IHM, afin d'être rapides à écrire. N'hésitez pas à ajouter vos propres idées ! J'indique dans chaque atelier quelques suggestions d'amélioration à faire, améliorations qui seront d'ailleurs peut-être implémentées dans les versions futures de mes applications : il est possible qu'à l'heure où vous lisiez ces lignes, la version sur le magasin (*store*) soit légèrement différente de celle présentée dans cette partie.

1. Créateur de couleurs

Difficulté : très facile
Plateformes ciblées : Windows 10, Android, iOS
Points techniques abordés : IHM, application multilingue.
Langages utilisés : JavaScript, HTML, CSS
Outils utilisés : Visual Studio, *Cordova*

Téléchargement de l'application :

- Pour Android : https://frama.link/J-qRFDbX
- Pour Windows 10 : https://frama.link/24H18DsZ
- Pour iOS : https://frama.link/cLKeNfRv

Téléchargement des codes sources : https://frama.link/0yz1bkwp

a) Description de l'application souhaitée

Nous avons souvent besoin, que l'on fasse de l'HTML/CSS ou un autre langage, d'utiliser une couleur à partir de ses composantes rouge, vert et bleu. Un petit outil permettant d'afficher une couleur en réglant les composantes est donc très pratique !

b) Conception de l'application



L'application est très simple ! Nous allons avoir un seul écran, pas de données « métier ».

L'écran contiendra une zone pour afficher un rectangle de couleur, une zone pour saisir ou lire la couleur sous la forme utilisée en HTML/CSS, et trois zones pour les composantes de la couleur (rouge, vert, bleu) permettant de les régler. Nous afficherons aussi la conversion des couleurs RVB en CMJN (Cyan, Magenta, Jaune, Noir) et en TSL (Teinte, Saturation, Luminosité) (26).

c) Développement : page HTML

L'écran en Cordova est une simple page HTML. Toute la page sera comprise dans une `div` de classe `app` (c'est le comportement par défaut). Un titre commencera la page, suivi d'une `div` pour la couleur.

Le titre sera localisable à l'aide de l'API `JQuery-Localize` (voir page 141) :

```
<h1 data-localize="title">RGBColor</h1>
<div id="color"></div>
```

Nous placerons ensuite la zone pour le code HTML de la couleur, composée d'un texte simple et d'une zone de saisie :

```
<div class="centre">
<label class="labelColor" for="htmlColor">HTML :</label>
<input class="word" type="text" id="htmlColor"
    value="#ffffff" /> </div>
```

Pour chaque composante de couleur, une div similaire sera utilisée, contenant un label, un slider, une zone de saisie. Exemple pour le rouge :

```
<div class="color">
    <label class="labelColor" for="red" data-
    localize="red">Red :</label>
    <input class="rangeColor" type="range" min="0" max="255"
    id="red" value="255" />
    <input class="byte" type="number" id="redVal" size="3"
    value="255" />
</div>
```

Les deux autres couleurs sont similaires.

Nous plaçons ensuite une div contenant les informations de teinte, saturation et luminosité :

```
<div id="hsl" class="groupbox">
    <h2 data-localize="hsl">Hue-Saturation-Luminance</h2>
    <input type="number" readonly id="hvalue" value="360" />
    <input type="number" readonly id="svalue" value="100" />
    <input type="number" readonly id="lvalue" value="100" />
</div>
```

Et enfin, une div contenant les informations CMJN :

```
<div id="cmjn" class="groupbox">
    <h2 data-localize="cmjn">Cyan-Magenta-Yellow-Black</h2>
    <input type="number" readonly id="cvalue" value="0" />
    <input type="number" readonly id="mvalue" value="0" />
    <input type="number" readonly id="jvalue" value="0" />
    <input type="number" readonly id="nvalue" value="0" />
</div>
```

Les chaînes sont quant à elles placées dans des fichiers JSON. Pour le français, le fichier s'appellera strings-fr.json et sera placé dans le sous-dossier res du dossier www :

```
{  "title": "Couleurs RVB",
  "red": "Rouge :",
  "green": "Vert :",
  "blue": "Bleu :",
  "hsl": "Teinte-Saturation-Luminance",
  "cmjn": "Cyan-Magenta-Jaune-Noir" }
```

De même un fichier `strings-en.json` contiendra la traduction en anglais, un fichier `strings-es.json` la traduction en espagnol, etc…

d) Développement : styles CSS

Vous pouvez régler les styles de manière à avoir un aspect plus agréable !

La zone pour la couleur va être centrée, occuper une grande partie de la largeur, mais nous lui donnerons une hauteur fixe. Une bordure fine noire sera plus jolie. Nous fixerons sa valeur comme *blanche* au départ :

```css
#color{
    width:80%;    height: 200px;
    margin-left:auto;
    margin-right:auto;
    margin-top:10px;
    margin-bottom:20px;
    border : 1px solid black;
    background-color:white;
}
```

Pour centrer un élément :

```css
.centre{
    text-align:center;
    margin-left:auto;
    margin-right:auto;
}
```

Chaque `label` de couleur va occuper 20% de la largeur :

```css
.labelColor{
    width:20%;
    display:inline-block;
    text-align:right;
}
```

Le `slider` occupera 60% de la largeur, et sera centré :

```css
.rangeColor{
    width:60%;
    vertical-align:middle;
}
```

La zone de saisie occupe le reste :

```css
.byte{     width:10%; }
```

Les deux div contenant les informations TSL et CMJN seront centrées avec une bordure noire :

```css
.groupbox {
    width: 80%;
    margin-left: auto;
    margin-right: auto;
    margin-top: 10px;
    border: 1px solid black;
    padding: 10px;
}
```

Les informations TSL, au nombre de 3, occuperont 30% de la largeur :

```css
#hsl input{
    width:30%;
    text-align:center; }
```

Les informations CMJN, elles, n'occuperont que 20% car elles sont au nombre de 4, et qu'il faut garder une petite marge :

```css
#cmjn input{
    width:20%;
    text-align:center;
}
```

e) Développement : partie code JavaScript

Commençons par nous occuper de la partie « traduction ». Au début du script index.js, une variable globale sera déclarée, elle servira à stocker les chaînes traduites :

```javascript
    var strings;
```

Dans la réponse à l'évènement onDeviceReady, nous allons utiliser JQuery-Localize pour modifier les chaînes traduites :

```javascript
$("[data-localize]").localize("res/strings", {
     callback: function (data,defaut) {
        strings = data;
        defaut(data);
     }
});
```

```
    function changeColor() {
        var red = $("#red").val();
        var green = $("#green").val();
        var blue = $("#blue").val();
        var rgb = (red << 16) | (green << 8) | blue;
```

Elle affiche ensuite la valeur « html » de cette couleur :

```
    var htmlColor = "#" + rgb.toString(16);
    $("#htmlColor").val(htmlColor);
```

Elle modifie la couleur de fond de la `div` nommée `color` :

```
document.getElementById("color").style.backgroundColor =
    htmlColor;
```

Elle modifie ensuite les zones de saisie des composantes :

```
$("#redVal").val(red);
$("#greenVal").val(green);
$("#blueVal").val(blue);
```

Il faut ensuite s'occuper de la conversion en TSL de la couleur (26). Une fois les composantes calculées, elles sont tout simplement transmises aux zones *ad hoc* :

```
var m1 = Math.max(red, green, blue);
var m2 = Math.min(red, green, blue);
var C = m1 - m2;
var T=0;
if (m1 === red && C!==0) {
    T = Math.round( 60 * (((green - blue) / C) % 6));
}
else if (m1 === green && C !== 0) {
    T = Math.round( 60 * (((blue - red) / C) +2 ));
}
else if (C !== 0) {
    T = Math.round(60 * (((red - green) / C) +4));
}
var S = Math.round(100 * C / m1);
var L = Math.round(100*(0.3 * red + 0.6 * green + 0.1 *
    blue)/255);
$("#hvalue").val(T.toString());
$("#svalue").val(S.toString());
$("#lvalue").val(L.toString());
```

Nous allons ensuite, toujours dans cette fonction, répondre à l'évènement change de chaque `slider`, à l'évènement `focusout` de chaque zone de saisie :

```
$(".rangeColor").on("change", changeColor);
$(".byte").on("focusout", changeText);
$("#htmlColor").on("focusout", changeHTML);
```

La fonction `changeText` est appelée à chaque fois que l'utilisateur modifie au clavier la valeur d'une composante de couleur. Elle changera la position du `slider` et la couleur globale :

```
    function changeText(input)      {
        var value = parseInt(input.target.value);
        if (value < 0) value = 0;
        if (value > 255) value = 255;
        input.target.value = value;
        var range = $(input.target.previousElementSibling);
        $(range).val(value);
        changeColor();
    }
```

La fonction `changeHTML` est appelée quand l'utilisateur modifie directement le code HTML (en hexadécimal) de la couleur. Cette opération va donc transformer les autres informations en conséquence :

```
function changeHTML() {
        var str = $("#htmlColor").val();
        str = str.substr(1, str.length - 1); // sans le #
        var rvb = parseInt(str, 16);
        var r = (rvb & 0xFF0000) >> 16;
        var g = (rvb & 0x00FF00) >> 8;
        var b = (rvb & 0x0000FF);
        $("#red").val(r);
        $("#redVal").val(r);
        $("#green").val(g);
        $("#greenVal").val(g);
        $("#blue").val(b);
        $("#blueVal").val(b);
        changeColor();
    }
```

La fonction `changeColor` est la plus importante. Son rôle est multiple, elle est appelée à chaque fois que la couleur est différente. Elle commence par calculer la couleur RVB à partir des `slider` :

```
    function changeColor() {
        var red = $("#red").val();
        var green = $("#green").val();
        var blue = $("#blue").val();
        var rgb = (red << 16) | (green << 8) | blue;
```

Elle affiche ensuite la valeur « html » de cette couleur :

```
    var htmlColor = "#" + rgb.toString(16);
    $("#htmlColor").val(htmlColor);
```

Elle modifie la couleur de fond de la `div` nommée `color` :

```
document.getElementById("color").style.backgroundColor =
     htmlColor;
```

Elle modifie ensuite les zones de saisie des composantes :

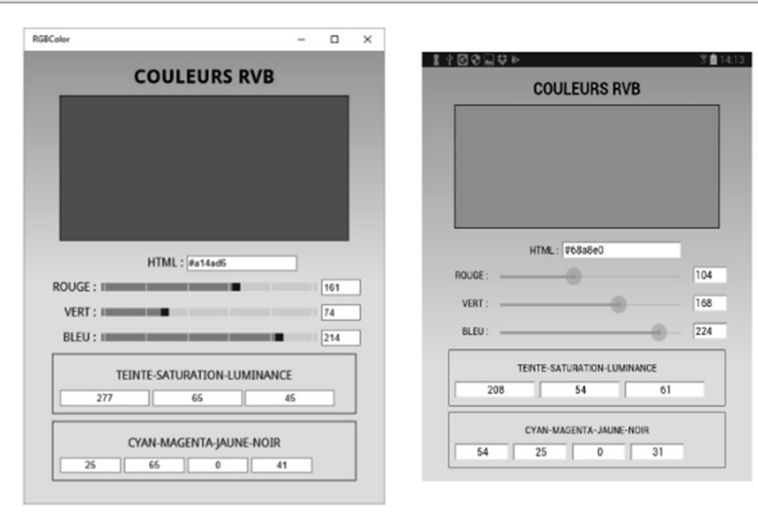
```
$("#redVal").val(red);
$("#greenVal").val(green);
$("#blueVal").val(blue);
```

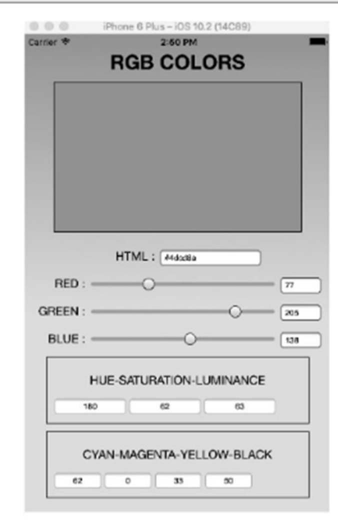
Il faut ensuite s'occuper de la conversion en TSL de la couleur (26). Une fois les composantes calculées, elles sont tout simplement transmises aux zones *ad hoc* :

```
var m1 = Math.max(red, green, blue);
var m2 = Math.min(red, green, blue);
var C = m1 - m2;
var T=0;
if (m1 === red && C!==0) {
     T = Math.round( 60 * (((green - blue) / C) % 6));
}
else if (m1 === green && C !== 0) {
     T = Math.round( 60 * (((blue - red) / C) +2 ));
}
else if (C !== 0) {
     T = Math.round(60 * (((red - green) / C) +4));
}
var S = Math.round(100 * C / m1);
var L = Math.round(100*(0.3 * red + 0.6 * green + 0.1 *
     blue)/255);
$("#hvalue").val(T.toString());
$("#svalue").val(S.toString());
$("#lvalue").val(L.toString());
```

Et enfin, la fonction se termine par la conversion en CMJN de la couleur RVB :

```
var N = Math.min(255 - red, 255 - green, 255 - blue);
C = Math.round(100 * (255 - red - N) / (255 - N));
var M = Math.round(100 * (255 - green - N) / (255 - N));
var J = Math.round(100 * (255 - blue - N) / (255 - N));
$("#cvalue").val(C.toString());
$("#mvalue").val(M.toString());
$("#jvalue").val(J.toString());
$("#nvalue").val(N.toString());
```





Capture 75 : application couleurs avec Cordova sous Windows, Android, iOS

f) Améliorations possibles

Cette application est volontairement très simple et limitée pour pouvoir être décrite totalement en peu de pages… La première version que j'ai déposée sur les différents magasins d'applications est celle décrite dans ces pages. Il est possible que je réalise des mises à jour de l'application, vous pouvez quant à vous essayer de l'améliorer en ajoutant des fonctionnalités, en modifiant l'IHM, etc.

Voici quelques idées de modifications intéressantes :

- Permettre le réglage (et non plus seulement la lecture) des valeurs TSL ; on peut imaginer un système de 3 *sliders*, sur le même principe que les *sliders* RVB, avec mise à jour de la couleur et des autres valeurs.
- Permettre le réglage des valeurs CMJN (même principe que ci-dessus). Pour limiter la taille de l'écran, il serait possible de

supprimer le rectangle de couleur et d'utiliser l'ensemble de l'écran pour afficher la couleur.

- Permettre la capture d'une couleur à l'aide de l'appareil photo (voir page 194) : une fois la photo prise, il faudra rechercher la couleur voulue puis déterminer ses composantes RVB et, ensuite, calculer les autres.

2. Scanner de QR Code

Difficulté : très facile

Plateformes ciblées : Windows 10, Android, iOS

Points techniques abordés : intégration de *plugins*, IHM simple, ressources

Langages utilisés : C#

Outils utilisés : Visual Studio, *Xamarin.Forms*

Téléchargement de l'application :

- Pour Android : https://frama.link/Sb1FxKhW
- Pour Windows 10 : https://frama.link/LBLfK4nq
- Pour iOS : https://frama.link/9vp5Sv_d

Téléchargement des codes sources : https://frama.link/xL5m8wAp

a) Description de l'application souhaitée

Les différents codes (code-barres, QR Code…) sont utilisés pour coder une information simple. Un téléphone équipé d'une caméra est très pratique pour « scanner » un code de ce type et le décoder.

Nous souhaitons donc réaliser une application de ce type, qui décodera un code (type QR code ou EAN) et réalisera ensuite les tâches suivantes :

- Si ce code est une URL, permettre de lancer le navigateur du téléphone pour accéder à celle-ci,
- Si ce code est un code-barres sur un produit alimentaire, permettre de lancer le navigateur sur un site donnant des informations nutritionnelles sur ce produit,

- Copier le texte décodé dans le presse-papiers[1] du téléphone dans les autres cas.

b) Conception de l'application

L'application ne réclame que peu de code métier, cependant elle peut être complexe sur la partie décodage du code… Pour nous faciliter la tâche, nous utiliserons une bibliothèque libre (*ZXing*) qui gère très bien le décodage d'un grand nombre de codes… Il ne reste donc plus grand-chose à concevoir à part la partie visuelle.

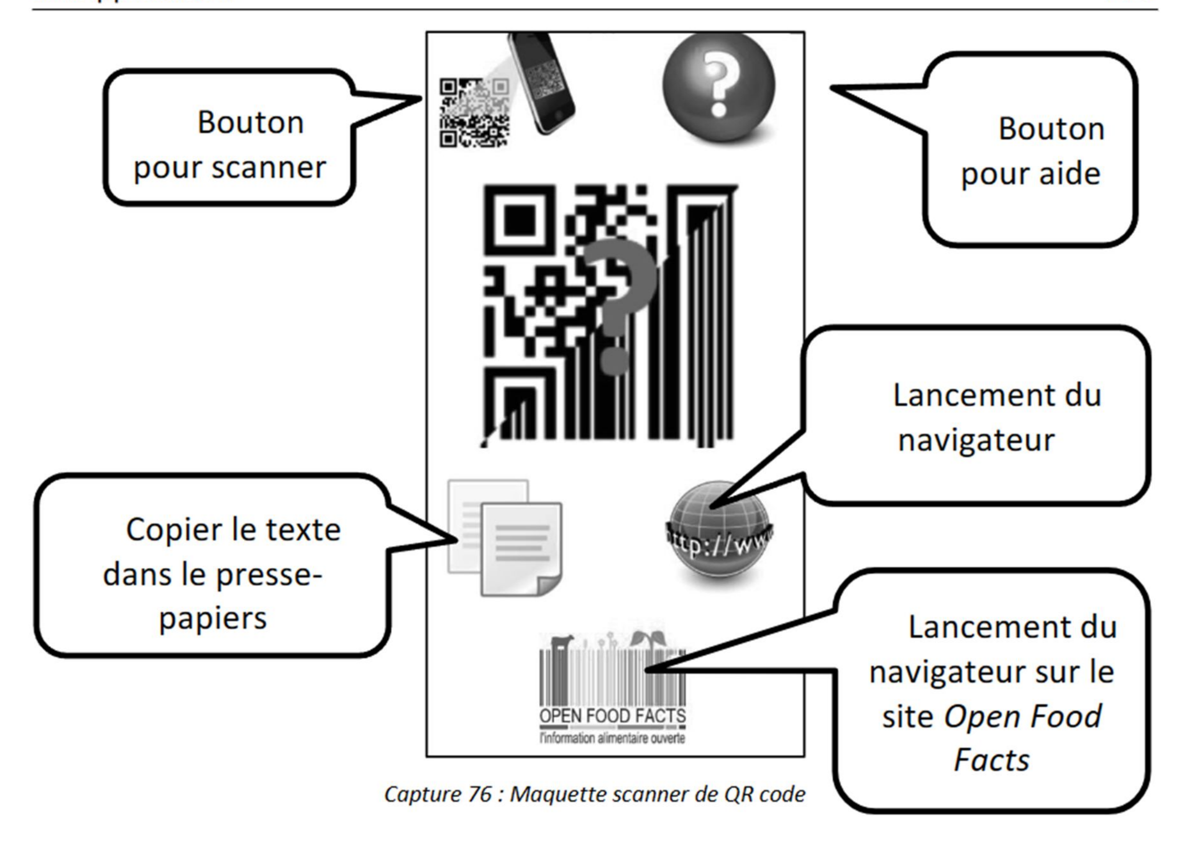
Notre écran sera volontairement très sobre et ne comprendra que des images (plus lisible que du texte, plus joli et de plus international). Nous avons besoin des boutons suivants :

- Lancer le *scan*
- Lancer le navigateur avec le résultat du *scan* (si URL)
- Lancer le navigateur sur le site *OpenFoodFacts* (si code-barres alimentaire)
- Copier le résultat du *scan* dans le presse-papiers
- Afficher une petite aide pour l'application.

Nous utiliserons en outre une petite image en centre de l'écran, image qui sera remplacée par le texte une fois le code scanné.

Pour la maquette suivante, j'ai utilisé des images sous licence CC trouvées sur Internet (je ne sais pas dessiner), à vous de choisir les images pour être clair, lisible et agréable à regarder.

[1] Le presse-papier est une zone de mémoire partagée du système qui stocke temporairement les objets (images, textes, etc.) qui sont « copiés » ou « coupés » en attendant d'être « collés ».

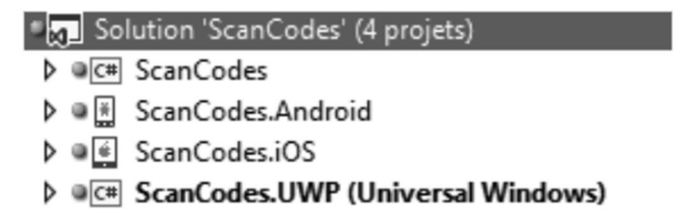


Capture 76 : Maquette scanner de QR code

c) Développement de l'application : départ

Nous allons utiliser le langage C#, l'éditeur Visual Studio et le *framework* *Xamarin.Forms* pour créer notre application sur les trois plateformes courantes.

Commençons donc par créer une application « *cross-platform* » avec Visual Studio en utilisant *Xamarin.Forms* pour les IHM. Comme d'habitude, le projet .NETStandard est plus intéressant que le projet partagé. Nous allons donc avoir une solution comprenant 4 projets comme sur la capture :

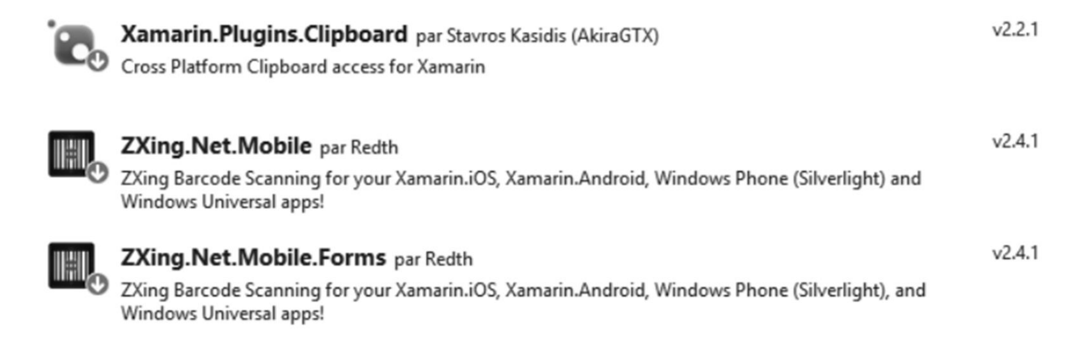


d) Développement : intégration des *plugins*

Il faut installer la bibliothèque *ZXing* qui sera chargée du décodage des codes, mais il nous faut également un *plugin* pour accéder au presse-papiers

de chaque plateforme (*Xamarin.Forms* ne prenant pas en charge la gestion du presse-papiers nativement).

Pour installer ces *plugins*, le plus simple est d'utiliser le gestionnaire *NuGet* de Visual Studio : clic-droit sur la solution, « gérer les packages *NuGet* pour la solution ». Choisir l'onglet « Parcourir » et saisir dans la zone de recherche le nom du paquet recherché. Installer le paquet dans les 4 projets de la solution. Réitérez l'opération pour les trois paquets suivants :



Capture 77 : paquets NuGet pour scanner de QR Code

e) Développement : création de l'écran

Tous les boutons étant des images, il faut placer des fichiers dans les ressources. Nous avons vu précédemment (au VI.3.c) deux méthodes pour intégrer des ressources images. Nous choisissons ici la méthode n'utilisant pas de code et nécessitant en revanche de placer chaque fichier image dans 3 projets (l'autre méthode permet de n'avoir le fichier image que dans un seul projet mais nécessite du code).

Pour le placement des contrôles, nous allons créer une grille avec 4 lignes et 2 colonnes. La ligne contenant l'image (puis le texte du code) étant plus grande que les autres.

```xml
<Grid.ColumnDefinitions>
     <ColumnDefinition Width="*"/>
     <ColumnDefinition Width="*"/>
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
     <RowDefinition Height="*"/>
     <RowDefinition Height="2*"/>
     <RowDefinition Height="*"/>
     <RowDefinition Height="*"/>
</Grid.RowDefinitions>
```

L'image d'accueil ainsi que le texte du code occupant le même espace, nous jouerons sur leur propriété `IsVisible` pour n'en afficher qu'un des deux. De même, les boutons pour utiliser le code ne seront actifs (propriété `IsEnabled`) que lorsque l'image sera décodée.

```xml
<Label x:Name="code" Grid.Row="1" Grid.ColumnSpan="2"
     IsVisible="False" HorizontalOptions="Center"
     VerticalOptions="Center" Margin="10"  />
<Image x:Name="image" Source="qr.png" Grid.Row="1"
     Grid.ColumnSpan="2" IsVisible="True"
     HorizontalOptions="FillAndExpand"
     VerticalOptions="FillAndExpand" Margin="10" />
```

Enfin, les boutons seront tous reliés à un fichier image :

```xml
<Button Image="Barcode_scanner.png" BorderRadius="10"
     Clicked="Scan" Grid.Row="0" Grid.Column="0"
     HorizontalOptions="FillAndExpand"
     VerticalOptions="FillAndExpand" Margin="10" />
<Button Image="help.png" Clicked="Help" BorderRadius="10"
     Grid.Row="0" Grid.Column="1" Margin="10"/>
<Button x:Name="site" Image="web.png" BorderRadius="10"
     Clicked="Go" Grid.Row="2" Grid.Column="1"
     IsEnabled="False" HorizontalOptions="FillAndExpand"
     VerticalOptions="FillAndExpand" Margin="10"  />
<Button x:Name="off" Clicked="off_Clicked" BorderRadius="10"
     Image="off.png" IsEnabled="False" Grid.Row="3"
     Grid.ColumnSpan="2" Margin="10" />
<Button x:Name="copy" Image="copy.png" Clicked="Copier"
     BorderRadius="10" IsEnabled="False" Grid.Row="2"
     Grid.Column="0" Margin="10"/>
```

f) Développement : décodage du code-barre

C'est la partie la plus complexe, mais c'est aussi celle que nous allons « délocaliser » à la bibliothèque *ZXing*.

Le *plugin* intégré comprend un certain nombre de classes, nous allons utiliser `ZXingScannerPage` qui est une page *Xamarin.Forms* intégrant directement la gestion de la caméra et de la reconnaissance de l'image. Cette page possède un évènement `OnScanResult` (appelé dès qu'un code a été trouvé) dans lequel nous allons arrêter le *scan* et gérer l'affichage. Attention : la modification de l'affichage doit être faite dans le *thread* principal, or la bibliothèque va utiliser un autre *thread*.

Nous allons donc créer une opération pour répondre au clic sur le bouton de *scan* :

```
private async void Scan(object sender, EventArgs args)
{
```

Dans un premier temps, le réglage des options puis la création de la page :

```
    var options = new MobileBarcodeScanningOptions
    {    TryHarder = true  };
    var scanPage = new ZXingScannerPage(options);
```

Il faut relier ensuite l'évènement `OnScanResult` de cette page à un code (ici créé directement). Dans ce code, commencez par arrêter le *scan* :

```
    scanPage.OnScanResult += (result) =>
    {
      scanPage.IsScanning = false;
```

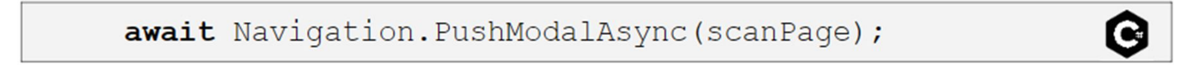
Ensuite, invoquez dans le *thread* principal le retour à la page précédente :

```
      Device.BeginInvokeOnMainThread(async () =>
      {
        try {
          await Navigation.PopModalAsync();
```

Le texte du code est récupéré, affiché dans le `Label` correspondant. Les propriétés `IsVisible` et `IsEnabled` sont ensuite mises à jour :

```
          code.Text = result.Text;
          image.IsVisible = false;
          code.IsVisible = true;
          if (result.BarcodeFormat ==
    ZXing.BarcodeFormat.QR_CODE &&
    code.Text.Contains("http"))
            site.IsEnabled = true;
          else if (result.BarcodeFormat ==
    ZXing.BarcodeFormat.EAN_13)
            off.IsEnabled = true;
          copy.IsEnabled = true;
        }
        catch {}
    });
```

Enfin, tout étant prêt, il faut passer à la page de *scan* :



```
    await Navigation.PushModalAsync(scanPage);
```

g) Suite du développement

L'application est (normalement) ergonomique, mais il est intéressant de donner à l'utilisateur une information sur l'utilisation de celle-ci. Nous allons la faire en deux langues (anglais et français) et donc utiliser les ressources textes. Pour l'affichage de l'aide, le code est très simple :

```
private void Help(object sender, EventArgs args)
{
     DisplayAlert(Strings.TitleHelp, Strings.Help,
     Strings.Ok);
}
```

Les trois chaînes sont issues des ressources (fichier `Strings.resx` pour la langue par défaut, l'anglais, et `Strings.fr.resx` pour le français : vous pouvez ajouter d'autres langues).

Pour lancer le navigateur par défaut du téléphone à l'adresse scannée (en général issue d'un QR Code), il suffit d'utiliser l'opération `OpenUri` de la classe `Device` :

```
private void Go(object sender, EventArgs args)
{
     Device.OpenUri(new Uri(code.Text));
}
```

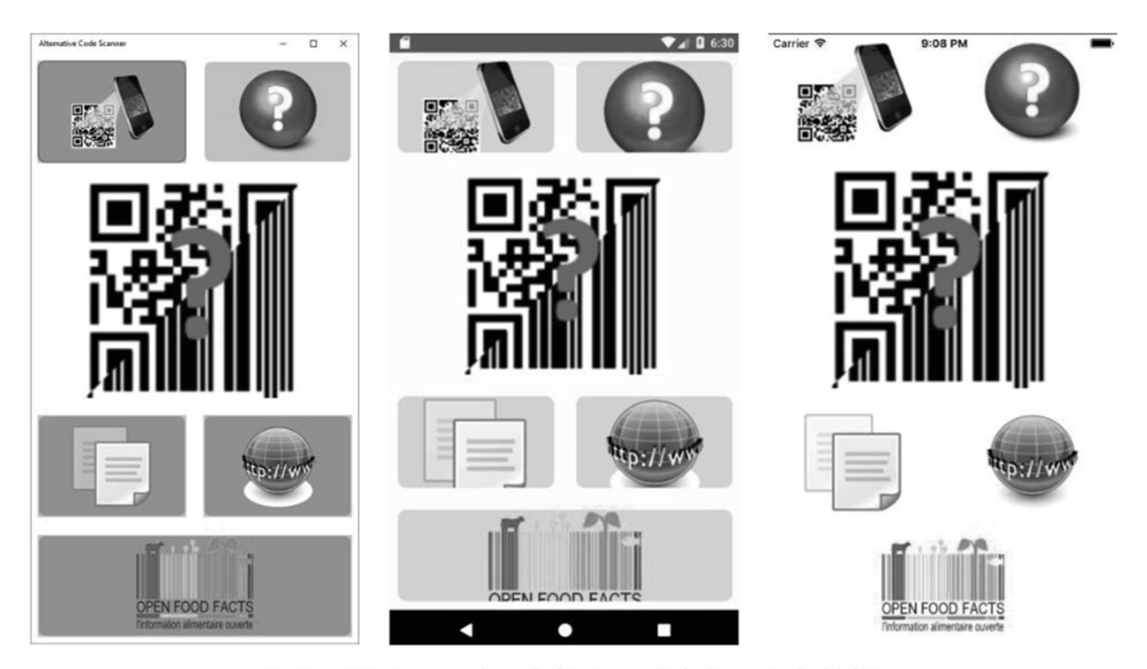
La copie dans le presse-papiers passe par l'utilisation d'un *plugin* mais reste très simple :

```
private void Copier(object sender, EventArgs args)
{
     CrossClipboard.Current.SetText(code.Text);
}
```

Enfin, pour aller sur le site *OpenFoodFacts* voir le produit référencé par le code-barres scanné (si celui-ci est un produit alimentaire) l'opération `OpenUri` sera là encore utilisée :

```
private void Off_Clicked(object sender, EventArgs e)
{
     String url = "https://fr.openfoodfacts.org/produit/" +
     code.Text;
     Device.OpenUri(new Uri(url));
}
```

L'application étant terminée, il est possible de la tester sur les différents émulateurs (attention certains ne prennent pas en compte la caméra) ou, mieux, directement sur un périphérique.



Capture 78 : Scanner de code-barres sur Windows, Android, iOS

h) Améliorations possibles

Cette application est volontairement très simple et limitée pour pouvoir être décrite totalement en peu de pages… La première version que j'ai déposée sur les différents magasins d'applications est celle décrite dans ces pages. Il est possible que je réalise des mises à jour de l'application, vous pouvez quant à vous essayer de l'améliorer en ajoutant des fonctionnalités, en modifiant l'IHM, etc…

Voici quelques idées de modifications intéressantes :

- Permettre le téléchargement d'un fichier au lieu de son ouverture dans le navigateur
- Remplacer le lien sur le site *OpenFoodFacts* par l'utilisation directe de son API dans l'application (voir chapitre X).

3. Minuteur de cuisine

Difficulté : facile

Plateformes ciblées : Android, iOS, Windows

Points techniques abordés : *timers*, alarmes, IHM simple, persistance

Langages utilisés : C#

Outils utilisés : Visual Studio, *Xamarin.Forms*

Téléchargement de l'application :

- Pour Android : https://frama.link/qz4yH35m
- Pour Windows : https://frama.link/wRSEA9Qv
- Pour iOS : https://frama.link/aA0wYyM1

Téléchargement des codes sources : https://frama.link/wyn377af

a) Description de l'application souhaitée

Pour réussir ses plats, qu'ils soient simples (moi je suis plutôt pâtes et œufs à la coque) ou compliqués, un minuteur est bien pratique. Ce minuteur devra stocker les durées précédentes afin d'être plus facile à utiliser et déclencher une alarme (active même si le téléphone est en veille).

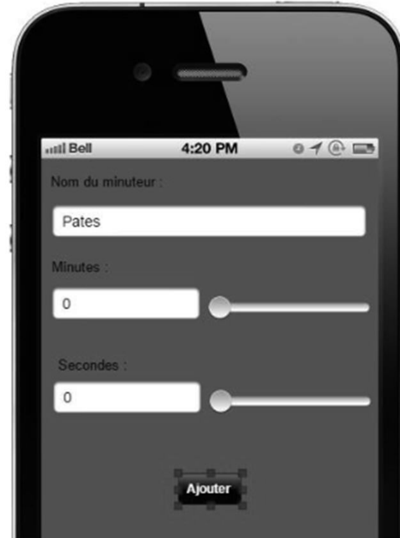
b) Conception de l'application : *design*

Un simple texte (assez gros) pour indiquer la durée restante du minuteur (minutes/secondes), suivi d'un petit texte (le nom du minuteur) ainsi qu'une série de boutons (démarrer le minuteur, le mettre en pause, le stopper, le remettre à zéro). Suit une liste contenant les minuteurs stockés et deux boutons pour ajouter ou enlever un minuteur.



Les boutons du haut se contentent de manipuler le minuteur, le bouton `enlever` de supprimer le minuteur sélectionné, mais le bouton `ajouter` va devoir ajouter un nouveau minuteur et donc ouvrir une nouvelle fenêtre pour régler celui-ci.

Pour régler la minuterie, il sera pratique de proposer soit la saisie numérique (clavier virtuel) soit un « glissé ». Le nom du minuteur lui servira d'identifiant.



c) Conception : couche métier

Le « métier » est ici simple : nous manipulons des minuteurs (un texte et une durée) ; le diagramme ci-dessous est donc à l'avenant :

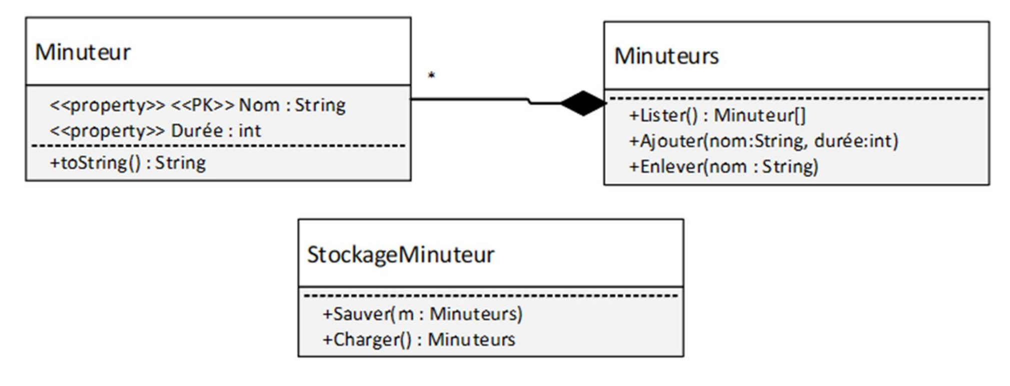


Diagramme 11 : couche métier application minuteur

d) Conception : couche IHM

Cette couche contient deux classes, correspondant aux deux fenêtres. Chaque classe sera associée avec la couche métier :

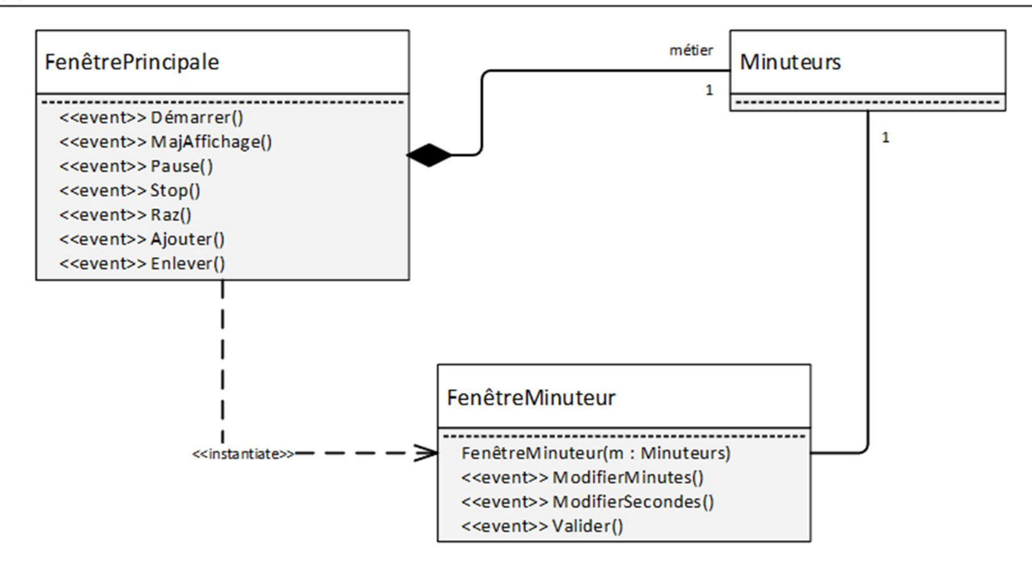


Diagramme 12 : couche IHM application minuteur

La partie supervisant l'alarme et la notification utilisateur étant gérée par le système, elle n'a pas besoin de « couche » particulière.

La fenêtre principale de l'application peut prendre les états suivants :

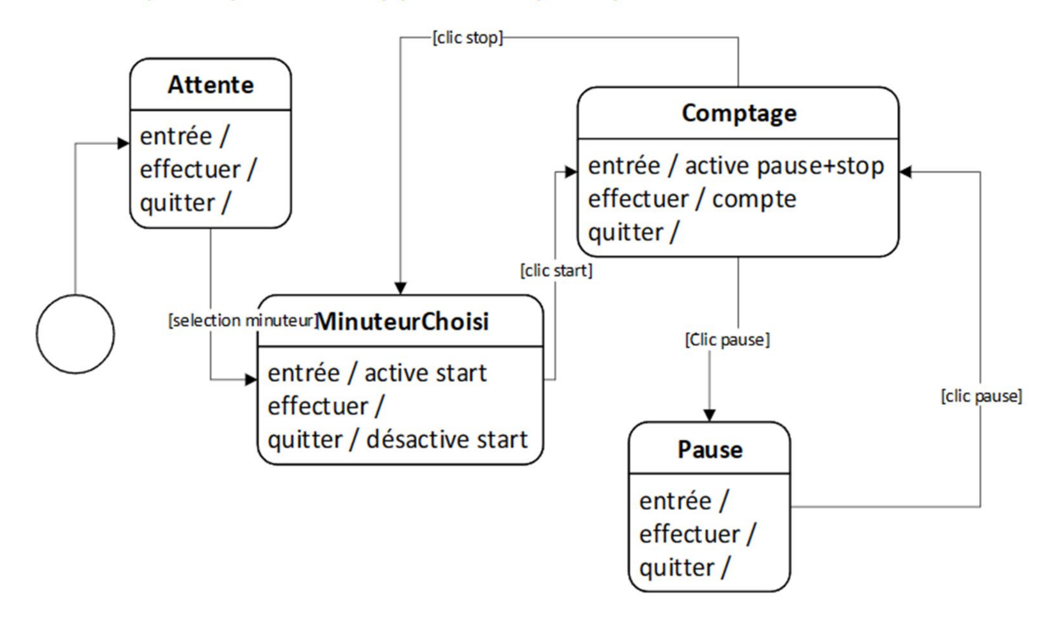


Diagramme 13 : états de l'application minuteur

e) Développement de l'application : métier

Il faut commencer par créer un projet d'application mobile multiplateforme, utilisant *Xamarin.Forms*.

Commençons par la classe `Minuteur`, assez simple ; tout d'abord il faut créer les propriétés :

```
class Minuteur
 {
     private string nom;
     private TimeSpan duree;

     public string Nom { get => nom; set => nom = value; }
     public TimeSpan Duree
    { get => duree; set => duree = value; }
```

Ensuite le constructeur :

```
public Minuteur(string nom, TimeSpan duree)
{
     this.nom = nom;
     this.duree = duree;
}
```

Et enfin, la méthode de conversion en chaîne :

```
public override string ToString()
{
     return nom + " (" + duree.ToString("mm:ss") + ")";
}
```

Passons à la classe `Minuteurs` qui compose la précédente : son seul attribut est la composition de `Minuteur` :

```
class Minuteurs
{
    private List<Minuteur> minuteurs = new List<Minuteur>();
```

L'opération `Lister` renvoie simplement un tableau :

```
        public Minuteur[] Lister()
        {
            return minuteurs.ToArray();
        }
```

`Ajouter` se contente de créer un minuteur et de l'ajouter à la liste :

```
        public void Ajouter(string nom, TimeSpan duree)
        {
            minuteurs.Add(new Minuteur(nom, duree));
        }
```

Pour `Enlever` il faut faire une recherche : le prédicat[1] de la fonction `RemoveAll` est ici bien pratique :

```
        public void Enlever(string nom)
        {
            minuteurs.RemoveAll((m) => m.Nom == nom);
        }
```

La couche métier se termine avec l'interface pour le stockage :

```
    interface IStockageMinuteur
    {
        void Sauver(Minuteurs m);
        Minuteurs Charger();
    }
```

Cette interface peut, pour des besoins de test, être implémentée par la classe suivante :

```
class StockageFactice : IStockageMinuteur
    {
        public Minuteurs Charger()
        {
            Minuteurs m = new Minuteurs();
            m.Ajouter("test", TimeSpan.FromSeconds(10));
            m.Ajouter("pâtes", TimeSpan.FromMinutes(10));
            m.Ajouter("riz", TimeSpan.FromMinutes(15));
            return m;
        }

        public void Sauver(Minuteurs m)
        {
            //
        }
    }
```

[1] Prédicat : expression dont l'évaluation produit un booléen. Voir fonctions Lambda (46).

f) Développement : écran principal

Nous allons utiliser une disposition de type grille pour réaliser l'écran. Quatre colonnes sont nécessaires, ainsi que cinq lignes (non égales) :

```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="*"/>
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
    <RowDefinition Height="2*"/>
    <RowDefinition Height="*"/>
    <RowDefinition Height="*"/>
    <RowDefinition Height="4*"/>
    <RowDefinition Height="*"/>
</Grid.RowDefinitions>
```

Des `Label` seront utilisés pour les textes, des `Button` pour les boutons et un contrôle `ListView` pour la liste des minuteurs :

```
<Label x:Name="time" Grid.Row="0" Grid.ColumnSpan="4"
    HorizontalOptions="CenterAndExpand"
    VerticalOptions="FillAndExpand" FontSize="48"
    Text="00:00" VerticalTextAlignment="Center"
    FontAttributes="Bold"/>
<Label x:Name="label" Grid.Row="1" Grid.ColumnSpan="4"
    HorizontalOptions="FillAndExpand"
    VerticalOptions="FillAndExpand" Text="..."
    HorizontalTextAlignment="Center"/>
<Button x:Name="start" IsEnabled="False" Grid.Row="2"
    Grid.Column="0" Margin="6" Text=">" Clicked="Démarrer"/>
<Button x:Name="pause" IsEnabled="False" Grid.Row="2"
    Grid.Column="1" Margin="6" Text="||" Clicked="Pauser"/>
<Button x:Name="stop" IsEnabled="False" Grid.Row="2"
    Grid.Column="2" Margin="6" Text="[]" Clicked="Stopper"/>
<Button x:Name="raz" IsEnabled="False" Grid.Row="2"
    Grid.Column="3" Margin="6" Text="0"/>
<ListView x:Name="liste" Grid.Row="3" Grid.ColumnSpan="4"
    Margin="10" ItemSelected="Selectionner" />
<Button Text="Ajouter" Grid.Row="4" Grid.Column="0"
    Grid.ColumnSpan="2" HorizontalOptions="FillAndExpand"
    Margin="10"/>
<Button Text="Enlever" Grid.Row="4" Grid.Column="2"
    Grid.ColumnSpan="2" HorizontalOptions="FillAndExpand"
    Margin="10"/>
```

Il faut ajouter, en attribut de la classe de fenêtre associée, un lien sur la couche métier et sur la couche de stockage :

```
private IStockageMinuteur stockage = new StockageFactice();
private Minuteurs minuteurs;
```

Initialisons ces liens dans le constructeur :

```
minuteurs = stockage.Charger();
```

Afin de mettre à jour l'affichage des minuteurs, le plus simple est de redéfinir l'opération `OnAppearing` de la page, appelée à chaque fois que la page s'ouvre :

```
protected override void OnAppearing()
{
    base.OnAppearing();
    Afficher();
}
```

Et enfin, l'opération pour afficher :

```
private void Afficher()
{
    liste.ItemsSource = null;
    liste.ItemsSource = minuteurs.Lister();
}
```

Occupons-nous à présent de l'affichage de la durée restante. Il nous faut un attribut (de type `TimeSpan`) qui va stocker la durée à afficher :

```
private TimeSpan durée;
```

Lors du démarrage du comptage, cette durée va être initialisée à partir du minuteur sélectionné. Un *timer* sera ensuite lancé : à chaque appel de celui-ci, la durée se décrémente de la durée écoulée entre deux « coups » du *timer* : il faut connaitre l'instant du « coup » précédent, dans un attribut du type `DateTime` :

```
private DateTime précédent;
```

A chaque appel du *timer*, il faut donc déduire le temps écoulé, et le décompter de la durée restante, puis afficher celle-ci, en renvoyant `true` si le *timer* doit continuer (il faut donc un booléen en attribut pour indiquer s'il faut le stopper ou non, ainsi qu'un attribut pour la pause) :

```
private bool TimerTick()
{
     if(!enPause)
     {
       TimeSpan ecart = DateTime.Now - précédent;
       durée -= ecart;
       time.Text = durée.ToString(@"mm\:ss");
     }

     précédent = DateTime.Now;
     return !stopper;
}
```

Le *timer* est lancé lors du clic sur le bouton *ad hoc* :

```
private void Démarrer(object sender, EventArgs args)
{
     if(liste.SelectedItem is Minuteur m)
     {
       précédent = DateTime.Now;
       durée = m.Duree;
       stopper = false;
       Device.StartTimer(TimeSpan.FromMilliseconds(500),
     TimerTick);
       start.IsEnabled = false;
       pause.IsEnabled = true;
       stop.IsEnabled = true;
     }
}
```

Le clic sur les boutons pour la pause et le stop est très simple et ne fait que mettre à jour l'état des boutons et du comptage :

```
private void Stopper(object sender, EventArgs args)
{
     stopper = true;
     pause.IsEnabled = false;
     start.IsEnabled = true;
     stop.IsEnabled = false;
}
private void Pauser(object sender, EventArgs args) {
     enPause = !enPause;
}
```

Lors de la sélection d'un minuteur, si aucun n'est lancé, il faut « préparer » :

```
private void Selectionner(object sender, EventArgs args)
{
     if (liste.SelectedItem is Minuteur m && stopper)
     {
       durée = m.Duree;
       time.Text = durée.ToString(@"mm\:ss");
       label.Text = m.Nom;
       start.IsEnabled = true;
     }
}
```

L'évènement `Clicked` du bouton `Enlever` est le suivant :

```
private void Enlever(object sender, EventArgs args)
{
     if(liste.SelectedItem is Minuteur m)
     {
       minuteurs.Enlever(m.Nom);
       Afficher();
     }
}
```

g) Développement : notification programmée

Pour la gestion de l'alarme, il faut utiliser une notification. Comme vu au IX.2.d), nous pouvons soit utiliser un *plugin* pour que le code soit dans le projet partagé, soit utiliser une interface avec un code spécifique pour chaque plateforme. C'est cette solution qui sera retenue, car elle évite l'installation du *plugin* qui, de plus, ne fonctionne pas bien quand le téléphone est en mode veille.

Dans le projet partagé, commun à toutes les plateformes, nous définissons notre interface et notre singleton pour contrôler la notification :

```
public interface INotifier
{
     void Envoyer(string titre, string message, DateTime
     when);
}
```

```
public class GereNotif {
     private static GereNotif instance = null;
     private GereNotif() { }
     public static GereNotif Instance
     {
        get
        {
           if (instance == null)
              instance = new GereNotif();
           return instance;
        }
     }
     public INotifier Notifier { get => notifier; set =>
     notifier = value; }

     private INotifier notifier;
     public void Envoyer(string titre, string message,
     DateTime time)
     {
        if (notifier != null)
           notifier.Envoyer(titre, message, time);
     }
}
```

A la fin de la fonction `Démarrer`, inclure le code suivant pour lancer la notification programmée :

```
DateTime time = DateTime.Now + durée;
GereNotif.Instance.Envoyer("Minuteur", "Durée terminée",
     time);
```

Dans le projet Windows (UWP), il faut ajouter la classe suivante, puis initialiser dans le constructeur de la page principale :

```
class UWPNotifier : INotifier {
     public void Envoyer(string titre, string message,
     DateTime when)
     {
        var content =
     ToastNotificationManager.GetTemplateContent(ToastTemplat
     eType.ToastImageAndText01);
        var lines = content.GetElementsByTagName("text");
        lines[0].AppendChild(content.CreateTextNode(message));
        ScheduledToastNotification toast = new
     ScheduledToastNotification(content, new
     DateTimeOffset(when));
        toast.Id = titre;
        var notifier =
```

```
    ToastNotificationManager.CreateToastNotifier();
      notifier.AddToSchedule(toast);
    }
}
GereNotif.Instance.Notifier = new UWPNotifier();
```

Il faut aussi bien insérer, dans le manifeste de l'application, les actifs visuels pour le « badge » et l'autoriser comme sur la capture suivante :

Notification de l'écran de verrouillage : Badge et texte de vignette

Dans le projet iOS, il faut ajouter la classe suivante :

```
class IOSNotifier : INotifier
{
    public IOSNotifier(UIApplication app)
    {
      if(UIDevice.CurrentDevice.CheckSystemVersion(8,0))
      {
        var settings =
    UIUserNotificationSettings.GetSettingsForTypes(UIUserNot
    ificationType.Alert | UIUserNotificationType.Badge |
    UIUserNotificationType.Sound,null);
        app.RegisterUserNotificationSettings(settings);
      }
    }

    public void Envoyer(string titre, string message,
    DateTime when)
    {
      UILocalNotification notification = new
    UILocalNotification
      {
        FireDate = NSDate.FromTimeIntervalSinceNow( (when-
    DateTime.Now).TotalSeconds),
        AlertAction = titre,
        AlertBody = message,
        ApplicationIconBadgeNumber = 1,
        TimeZone = NSTimeZone.DefaultTimeZone,
        SoundName = UILocalNotification.DefaultSoundName
      };
    UIApplication.SharedApplication.ScheduleLocalNotificatio
    n(notification);
    }
}
```

Il faut également l'opération suivante dans la classe `AppDelegate` :

```
public override void ReceivedLocalNotification(UIApplication
     application, UILocalNotification notification)
{
     UIAlertController okayAlertController =
     UIAlertController.Create(notification.AlertAction,
     notification.AlertBody, UIAlertControllerStyle.Alert);
     okayAlertController.AddAction(UIAlertAction.Create("OK",
     UIAlertActionStyle.Default, null));
     UIApplication.SharedApplication.KeyWindow.RootViewContro
     ller.PresentViewController(okayAlertController, true,
     null);
     UIApplication.SharedApplication.ApplicationIconBadgeNumb
     er = 0;
}
```

Toujours dans la classe `AppDelegate`, il ne reste qu'à initialiser la classe à la fin de l'opération `FinishedLaunching` :

```
GereNotif.Instance.Notifier = new IOSNotifier(app);
```

Et enfin, dans le projet Android, créez la classe suivante :

```
class AndroidNotifier : BroadcastReceiver, INotifier
{
     private Context context;
     private string titre;
     private string message;
     public AndroidNotifier(Context context)
     {
       this.context = context;
     }
     public void Envoyer(string titre, string message,
     DateTime when)
     {
       this.titre = titre;
       this.message = message;
       AlarmManager am =
     context.GetSystemService(Context.AlarmService) as
     AlarmManager;
       context.RegisterReceiver(this, new
     IntentFilter("CookTimer"));
       PendingIntent pi = PendingIntent.GetBroadcast(context,
     0, new Intent("CookTimer"), 0);
       DateTime baseTime = new DateTime(1970, 1, 1, 0, 0, 0,
     DateTimeKind.Utc);
       TimeSpan span = when.ToUniversalTime() - baseTime;
       am.Set(AlarmType.RtcWakeup,
     (long)span.TotalMilliseconds, pi);
```

```
    }
    public override void OnReceive(Context context, Intent
    intent)
    {
      NotificationManager manager =
    context.GetSystemService(Context.NotificationService) as
    NotificationManager;
      PendingIntent pi = PendingIntent.GetActivity(context,
    0, new Intent(), PendingIntentFlags.OneShot);
      Notification.Builder builder = new
    Notification.Builder(context);
      builder.SetTicker("CookTimer");
      builder.SetContentTitle(titre);
      builder.SetContentText(message);
      builder.SetSmallIcon(Resource.Drawable.badge);

      builder.SetSound(RingtoneManager.GetDefaultUri(Rington
    eType.Notification));
      manager.Notify(100, builder.Build());
    }
}
```

Initialiser enfin à la fin de l'opération `OnCreate` de l'activité principale :

```
GereNotif.Instance.Notifier = new AndroidNotifier(this);
```



h) Développement : écran de réglage

Créez dans le projet partagé une deuxième page. Cette page va contenir une disposition en grille de deux colonnes et sept lignes :

```
<Grid.ColumnDefinitions><ColumnDefinition Width="*"/>
    <ColumnDefinition Width="*"/></Grid.ColumnDefinitions>
<Grid.RowDefinitions>
    <RowDefinition Height="*"/>
    <RowDefinition Height="*"/>
    <RowDefinition Height="*"/>
    <RowDefinition Height="*"/>
    <RowDefinition Height="*"/>
    <RowDefinition Height="*"/>
    <RowDefinition Height="*"/>
</Grid.RowDefinitions>
```

Les contrôles sont ensuite placés suivant le modèle vu page 279 :

```
<Label Grid.Row="0" Text="Nom du minuteur :"
    VerticalOptions="End" FontSize="Medium" />
```

```
<Label Grid.Row="2" Text="Minutes :" VerticalOptions="End"
     FontSize="Medium"/>
<Label Grid.Row="4" Text="Secondes :" VerticalOptions="End"
     FontSize="Medium" />
<Entry Grid.Row="1" Grid.ColumnSpan="2" Margin="10"
     x:Name="nom" Placeholder="nom du minuteur"
     VerticalOptions="Center" FontSize="Medium" />
<Entry Grid.Row="3" Grid.Column="0" Margin="10"
     x:Name="minutes" Placeholder="min" Keyboard="Numeric"
     VerticalOptions="Center" FontSize="Medium"/>
<Entry Grid.Row="5" Grid.Column="0" Margin="10"
     x:Name="secondes" Placeholder="sec" Keyboard="Numeric"
     VerticalOptions="Center" FontSize="Medium"/>
<Slider Grid.Row="3" Grid.Column="1" Margin="5"
     x:Name="sliderMin" Minimum="0" Maximum="59"
     VerticalOptions="Center" ValueChanged="ChangeMin" />
<Slider Grid.Row="5" Grid.Column="1" Margin="5"
     x:Name="sliderSec" Minimum="0" Maximum="59"
     VerticalOptions="Center" ValueChanged="ChangeSec" />
<Button Grid.Row="6" Grid.ColumnSpan="2"
     HorizontalOptions="Center" VerticalOptions="Center"
     Text="Ajouter" FontSize="Medium" Clicked="Ajouter" />
```

Cette page va nécessiter un lien sur une instance de `Minuteurs`, initialisée dans le constructeur :

```
private Minuteurs modèle;
public SettingsPage (Minuteurs minuteurs)
{
     InitializeComponent ();
     modèle = minuteurs;
}
```

Nous allons utiliser les évènements `ValueChanged` des deux `Slider` pour mettre à jour les zones de saisie :

```
private void ChangeMin(object sender, EventArgs args)
{
     minutes.Text = sliderMin.Value.ToString("0");
}
private void ChangeSec(object sender, EventArgs args)
{
     secondes.Text = sliderSec.Value.ToString("0");
}
```

Le clic sur le bouton `ajouter` provoquera, lui, l'ajout d'un `Minuteur` au modèle et le retour à la page précédente :

```
private void Ajouter(object sender, EventArgs args)
{
    try
    {
      int m = Convert.ToInt32(minutes.Text);
      int s = Convert.ToInt32(secondes.Text);
      modèle.Ajouter(nom.Text, new TimeSpan(0, m, s));
      Navigation.PopModalAsync();
    }
    catch(Exception x)
    {
      DisplayAlert("Erreur", x.Message, "Ok");
    }
}
```

Il ne reste plus qu'à écrire l'évènement `Clicked` du bouton `Ajouter` dans la page principale :

```
private async void Ajouter(object sender, EventArgs args)
{
    await Navigation.PushModalAsync(new
    SettingsPage(minuteurs));
}
```

L'application est testable et devrait fonctionner sur les trois systèmes. Seule manque la persistance des données.

i) Développement : persistance des données

Les seules données à stocker sont les minuteurs. Le moyen le plus simple est de recourir à la sauvegarde par paramètres (voir VII.1.d) et la sérialisation.

Xamarin.Forms utilisant *.NETStandard*, il faut que celui-ci supporte la sérialisation. C'est le cas depuis la version 1.3, mais le paquet nécessaire est inclus par défaut depuis la version 2.0 uniquement. Regardez la version *.NETStandard* ciblée (clic droit sur le projet partagé → propriétés) : si celle-ci est inférieure à 1.3, augmentez-la à 1.4 ou plus et installez les *packages NuGet* pour la sérialisation :

- `System.Runtime.Serialization.Primitives`
- `System.Runtime.Serialization.Xml`

Si la version de *.NETStandard* est 2.0 ou plus vous n'avez rien d'autre à faire.

Dans tous les cas, pour pouvoir utiliser les paramètres de manière multiplateforme, il faut installer le package *NuGet* `Xam.Plugin.Settings` déjà vu.

Une fois les installations faites, il faut marquer par `[DataContract]` les classes métier (`Minuteur` et `Minuteurs`) et marquer leurs attributs par `[DataMember]`.

Ajoutez une nouvelle classe dans le projet partagé :

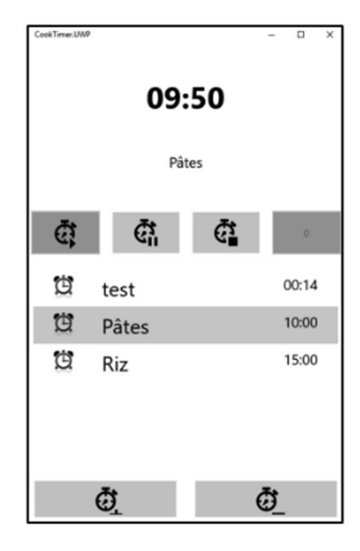
```
public class StockageParams : IStockageMinuteur {
    public Minuteurs Charger() {
      Minuteurs m = null;
      try    {
      string
    s=CrossSettings.Current.GetValueOrDefault("minuteurs",
    "");
        if (s != ""){
          using (MemoryStream flux = new
    MemoryStream(Encoding.UTF8.GetBytes(s)))   {
            DataContractSerializer ser = new
    DataContractSerializer(typeof(Minuteurs));
            m = ser.ReadObject(flux) as Minuteurs; }}
        else m = new Minuteurs();
      }
      catch  {
        m = new Minuteurs();
      }
      return m;
    }

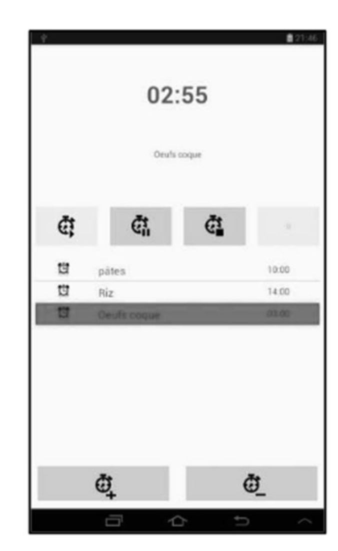
    public void Sauver(Minuteurs m){
      DataContractSerializer ser = new
    DataContractSerializer(typeof(Minuteurs));
      using (MemoryStream flux = new MemoryStream())      {
        ser.WriteObject(flux, m);
        string s = Encoding.UTF8.GetString(flux.ToArray(),
    0,(int) flux.Length);
        CrossSettings.Current.AddOrUpdateValue("minuteurs",
    s);
      }
    }
}
```

Dans la page principale, remplacez l'utilisation du stockage factice par cette classe :

```
private IStockageMinuteur stockage = new StockageParams();
```


L'application est à présent fonctionnelle, elle peut être agrémentée de quelques images :







Capture 79: minuteur de cuisine sous Windows, Android, iOS

j) Améliorations possibles

Cette application est volontairement très simple et limitée pour pouvoir être décrite totalement en peu de pages… La première version que j'ai déposée sur les différents magasins d'applications est celle décrite dans ces pages. Il est possible que je réalise des mises à jour de l'application, vous pouvez quant à vous essayer de l'améliorer en ajoutant des fonctionnalités, en modifiant l'IHM, etc…

Voici quelques idées de modifications intéressantes :

- Une IHM plus attrayante à l'œil
- La possibilité de lancer plusieurs alarmes en même temps

4. Capteurs

Difficulté : facile

Plateformes ciblées : Windows 10, Android, iOS

Points techniques abordés : utilisation des capteurs

Langages utilisés : C++

Outils utilisés : Qt Creator

Téléchargement de l'application :

- Pour Android : https://frama.link/LVwM52jd
- Pour Windows 10 : https://frama.link/XKB25zfw
- Pour iOS : https://frama.link/7YNdE4JL

Téléchargement des codes sources : https://frama.link/f95vwc2o

a) Descriptif de l'application souhaitée

Cet atelier a pour but de développer une petite application pour téléphone permettant d'utiliser les différents capteurs physiques pour connaitre l'environnement de l'appareil. L'application sera une sorte de « couteau suisse » avec diverses fonctionnalités (simples), notamment :

- Détermination de la position (coordonnées géophysiques)
- Vitesse instantanée
- Niveau (déterminer si le téléphone est sur une surface plane)
- Boussole (déterminer le nord magnétique)

L'atelier est une application directe du chapitre 0 : à lire avant !

b) Conception de l'application : *design*

L'application contiendra plusieurs écrans :

- Ecran d'accueil avec un menu de choix
- Ecran « GPS » avec la position courante en temps réel et la vitesse instantanée
- Ecran « boussole » avec une image de boussole mise à jour en temps réel
- Ecran « niveau » avec une visualisation du niveau de l'appareil
- Ecran « distance » avec la possibilité de calculer une distance entre un point de départ et un point d'arrivée

Afin d'améliorer l'affichage, j'ai utilisé des icônes gratuites trouvées sur (27).

Voici le *design* des écrans de l'application :



Figure 7 : design de l'application capteurs

c) Conception de l'application : architecture

L'application ne manipule pas de données, n'interagit pas avec un site, se contente d'utiliser les API du système sur les capteurs : il n'y a en fait qu'une seule couche applicative, la couche IHM. Dans cette couche, il y aura une classe par écran/fenêtre/activité. Le diagramme de classes est donc très simple :

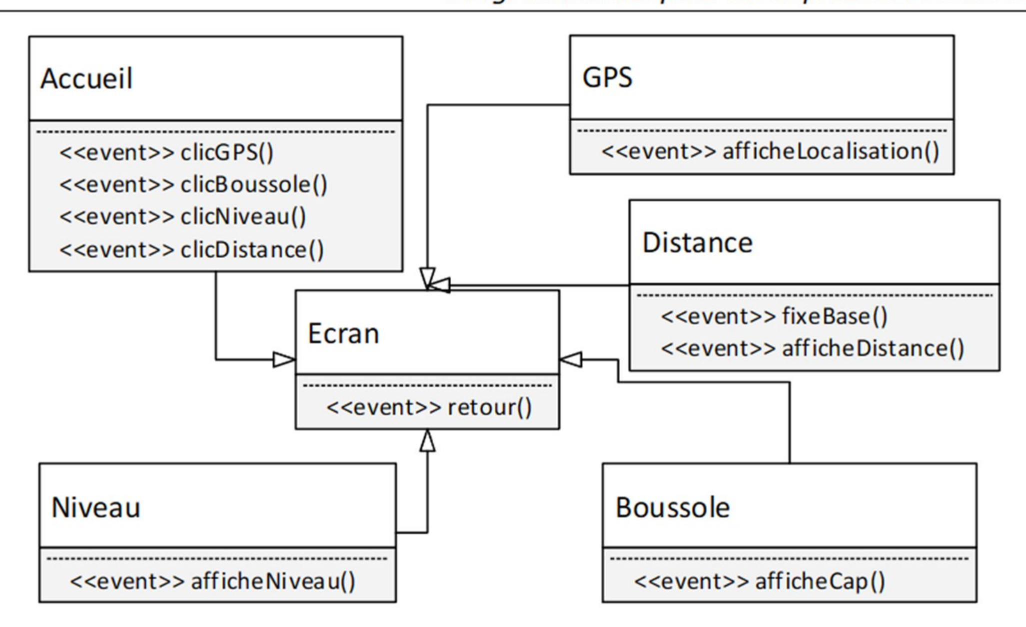


Diagramme 14 : conception application capteurs

d) **Développement de l'application : départ**

L'application Qt est basée sur cinq fenêtres : la fenêtre d'accueil, la fenêtre pour le GPS, celle pour la boussole, celle pour le niveau et celle pour la mesure de distance.

Il faut commencer par placer dans les ressources Qt (voir VI.3.c)) les images nécessaires à l'application.

Pour la première fenêtre, il suffit de placer quatre composants `ToolButton` et de configurer leur propriété `Icon` à l'une des images ressources liées.



La fenêtre suivante servira pour le GPS, elle va utiliser un `layout` de type formulaire, avec un `Label` pour l'image, des `Label` pour le texte et des `LineEdit` pour l'affichage des valeurs.

e) Développement : lecture du GPS

Le code pour l'affichage des informations GPS est similaire à celui du chapitre VIII.1.b).

Il faut une opération pour afficher correctement les coordonnées GPS :

```
QString VisuGPS::convertGPS(double angle, QString str) {
    QString retour;
    int pos;
    if(angle<0){pos = 1; angle=-angle;  }
    else pos=0;
    int degres = (int)angle;
    int minutes = (angle-degres)*60;
    int secondes = ((angle-degres)*60-minutes)*60;
    retour = QString::number(degres)+"°
     "+QString::number(minutes)+"'
     "+QString::number(secondes)+"\" "+str[pos];
    return retour;
}
```

Cette fonction sera utilisée dans le *slot* suivant :

```
void VisuGPS::nouvellePosition(const QGeoPositionInfo &info)
{
    double altitude = info.coordinate().altitude();
    double longitude = info.coordinate().longitude();
    double latitude = info.coordinate().latitude();

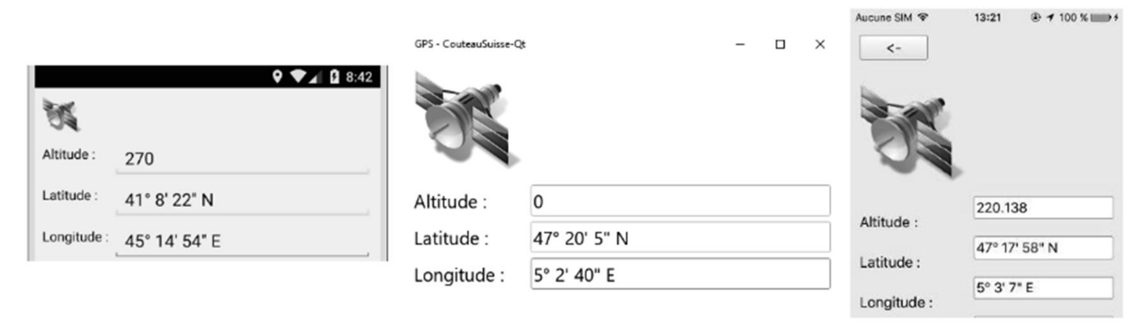
    ui->latitude->setText(convertGPS(latitude,"NS"));
    ui->longitude->setText(convertGPS(longitude,"EO"));
    ui->altitude->setText(QString::number(altitude));
}
```

A la création de la fenêtre, initialisez le travail :

```
VisuGPS::VisuGPS(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::VisuGPS)
{
    ui->setupUi(this);
    this->setWindowState(Qt::WindowFullScreen);
    source =
     QGeoPositionInfoSource::createDefaultSource(this);
    if(source)
    {
        connect(source,
     SIGNAL(positionUpdated(QGeoPositionInfo)),this,
     SLOT(nouvellePosition(QGeoPositionInfo)));
        source->startUpdates();
    }
    else
    {
        QMessageBox::warning(this,"Erreur",
          "Pas de capteur de position");
    }
}
```

Dans la fenêtre principale, reliez au signal « *clicked* » du bouton GPS le *slot* suivant :

```
    VisuGPS fiche(this);
    fiche.exec();
```



Capture 80 : GPS avec Qt sous Android, Windows, iOS

Sous iOS il n'y a pas de bouton « retour » et il est nécessaire de créer un bouton cliquable pour cela. Nous ajoutons donc le bouton pour fermer la fenêtre.

f) Développement : la boussole

La fenêtre contient deux `QLabel` : un pour une image décorative (taille fixe), l'autre pour la boussole qui indiquera le nord (taille maximisée).

Dans la classe de fenêtre, déclarez un attribut pour la boussole elle-même, ainsi qu'un *slot* pour la mise à jour du capteur :



```
private:
    QCompass *boussole;
private slots:
    void update();
```

Dans le constructeur, initialisez la boussole, relier le *slot* au signal émis par le capteur et lancer celui-ci :

```
boussole = new QCompass(this);
boussole->setDataRate(1);
connect(boussole,SIGNAL(readingChanged()),this,SLOT(update())
    );
boussole->start();
```

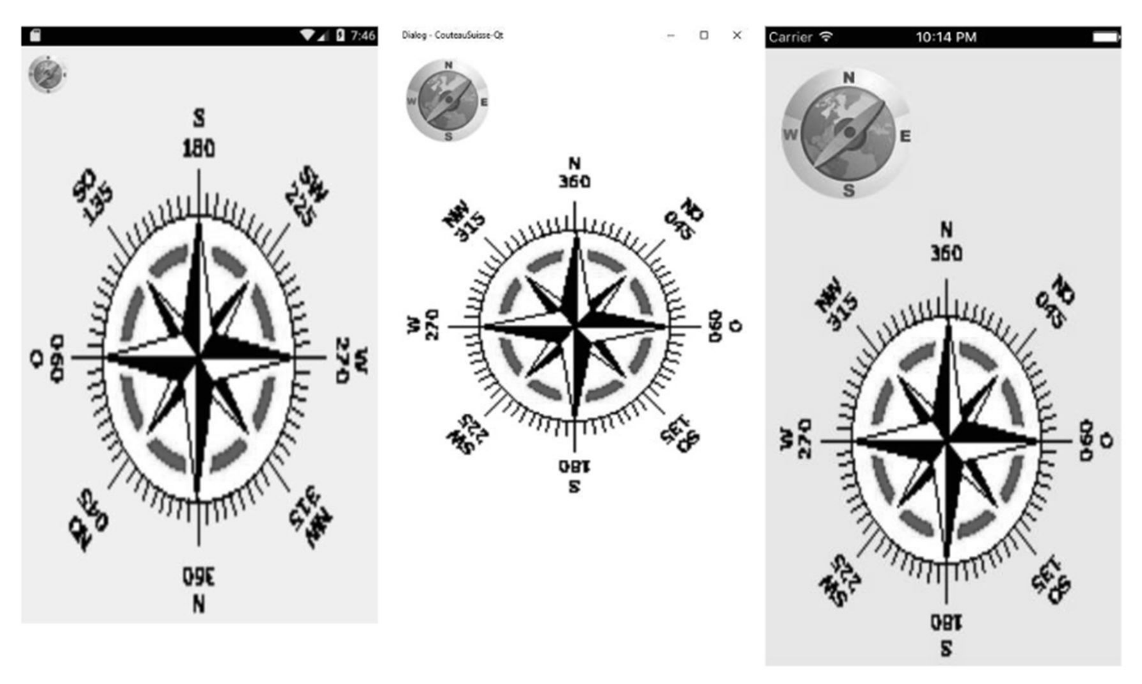
Dans le destructeur, libérez la mémoire allouée :

```
delete boussole;
```

A intervalle de temps régulier, la boussole va donc appeler cette opération et appliquer une rotation à l'image :

```cpp
void Boussole::update()
{
    QCompassReading* read = boussole->reading();
    if(read)
    {
        qreal angle = read->azimuth();
        int width = ui->label->pixmap()->width();
        int height = ui->label->pixmap()->height();
        QTransform trans;
        trans = trans.translate(width/2, height/2);
        trans = trans.rotate(angle);
        QPixmap bmp(":/icones/img/boussole2.png");
        QPixmap map = bmp.transformed(trans);
        ui->label->setPixmap(map);
    }
}
```

Cette partie est testable dans un téléphone (s'il possède un magnétomètre) ou dans un émulateur.



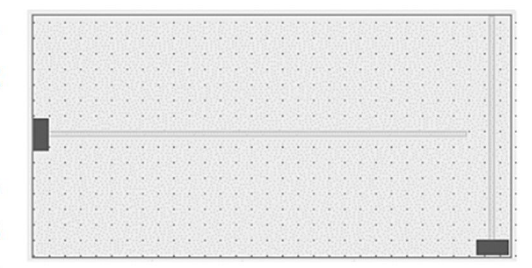
Capture 81 : boussole avec Qt sous Android, Windows, iOS

g) Développement : le niveau

Un niveau utilise en général des bulles d'air dans un liquide pour déterminer l'horizontalité d'une surface. Nous allons quant à nous utiliser les capteurs de notre téléphone, notamment le capteur de rotation.



Commençons par placer dans la fenêtre une zone (*layout* horizontal) en haut contenant le bouton de retour et l'icône du niveau :



En bas de la fenêtre, plaçons une zone (*layout* horizontal) contenant deux *sliders* horizontal et vertical :

Le capteur de rotation donnant les angles en degrés, nous allons définir les valeurs minimales et maximales de ces *sliders* respectivement à -180 et 180. Un angle de 0 correspondra donc à la position médiane de chaque *slider*.

Ajoutons à la classe de fenêtre un attribut pour le capteur de rotation, ainsi qu'un *slot* pour la lecture :

```cpp
private slots:
    void lecture();
private:
    QRotationSensor *sensor;
```

Cette variable est initialisée dans le constructeur de la fenêtre, et libérée dans son destructeur :

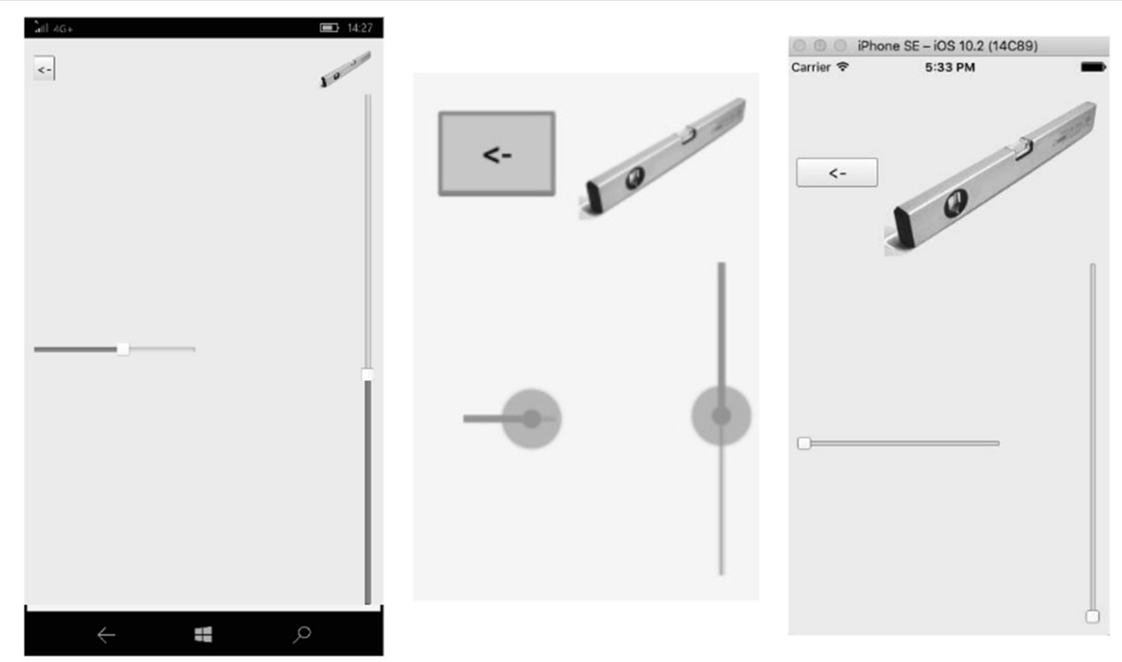
```cpp
niveau::niveau(QWidget *parent) : QDialog(parent),
      ui(new Ui::niveau)
{
    ui->setupUi(this);
    sensor = new QRotationSensor(this);

     connect(sensor,SIGNAL(readingChanged()),this,SLOT(lectur
     e())) ;
     sensor->setDataRate(10);
    if(!sensor->start()) {
        QMessageBox::warning(this,"Erreur",
          "Erreur avec le capteur de rotation");
```

```
    }
}
niveau::~niveau() {
    delete sensor;
    delete ui;
}
```

La lecture du capteur de rotation mettra à jour la position de chaque *slider* en fonction des axes de rotation en X et en Y :

```
void niveau::lecture()
{
    QRotationReading *reading = sensor->reading();
    ui->xSlider->setValue(reading->x());
    ui->ySlider->setValue(reading->y());
}
```

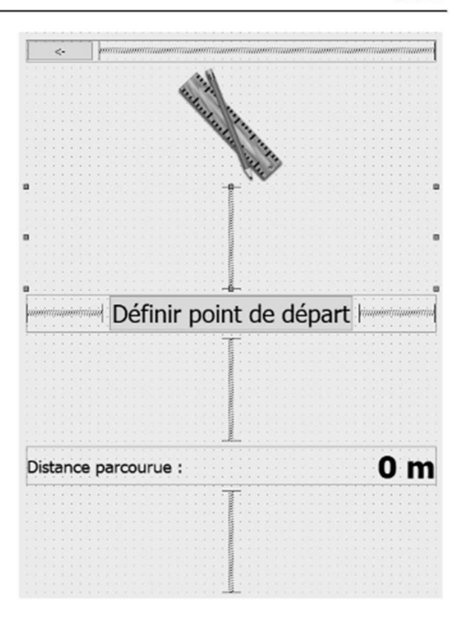


Capture 82 : capteurs de niveau avec Qt sous Windows, Android, iOS

h) Développement : la mesure de la distance

La fenêtre comprendra :

- Un bouton pour le retour,
- Un *label* pour l'image décorative,
- Un bouton pour fixer le point de départ,
- Un *label* pour indiquer la distance parcourue.



Afin de garantir une disposition correcte, la page sera disposée en *layout* vertical, des *layout* horizontaux seront utilisés ainsi que des *spacer* pour améliorer l'aspect, comme sur la capture :

Afin de calculer la distance, la page utilisera le capteur de position déjà utilisé au XII.4.e).

Pour calculer une distance, il faut faire la différence entre deux positions. Nous aurons donc besoin dans notre fenêtre, comme attributs :

- D'un pointeur sur une source de géolocalisation
- D'une position courante
- D'une position « de base »

```
    QGeoPositionInfoSource* source;
    QGeoCoordinate courante;
    QGeoCoordinate base;
```

Il faut aussi un *slot* pour l'associer à la source de géolocalisation, ainsi qu'un *slot* pour répondre au clic sur le bouton « Définir le point de départ » :

```
private slots:
    void nouvellePosition(const QGeoPositionInfo& info);
    void on_pushButton_2_clicked();
```

Dans le constructeur, initialisez le tout, exactement comme dans la fenêtre de visualisation des coordonnées GPS :

```
ui->setupUi(this);
this->setWindowState(Qt::WindowMaximized);
source = QGeoPositionInfoSource::createDefaultSource(this);
if(source) {
     connect(source,
        SIGNAL(positionUpdated(QGeoPositionInfo)),this,
        SLOT(nouvellePosition(QGeoPositionInfo)));
     source->setUpdateInterval(500);
     source->setPreferredPositioningMethods(
        QGeoPositionInfoSource::AllPositioningMethods);
     source->startUpdates();
```

```
}
else
{
     QMessageBox::warning(this,"Erreur","Pas de capteur de
     position");
}
```

De même, le destructeur libère la mémoire :

```
    delete ui;
    delete source;
```

Le *slot* associé à la source de géolocalisation se contente de stocker la position actuelle et d'afficher la distance calculée, via une opération privée :

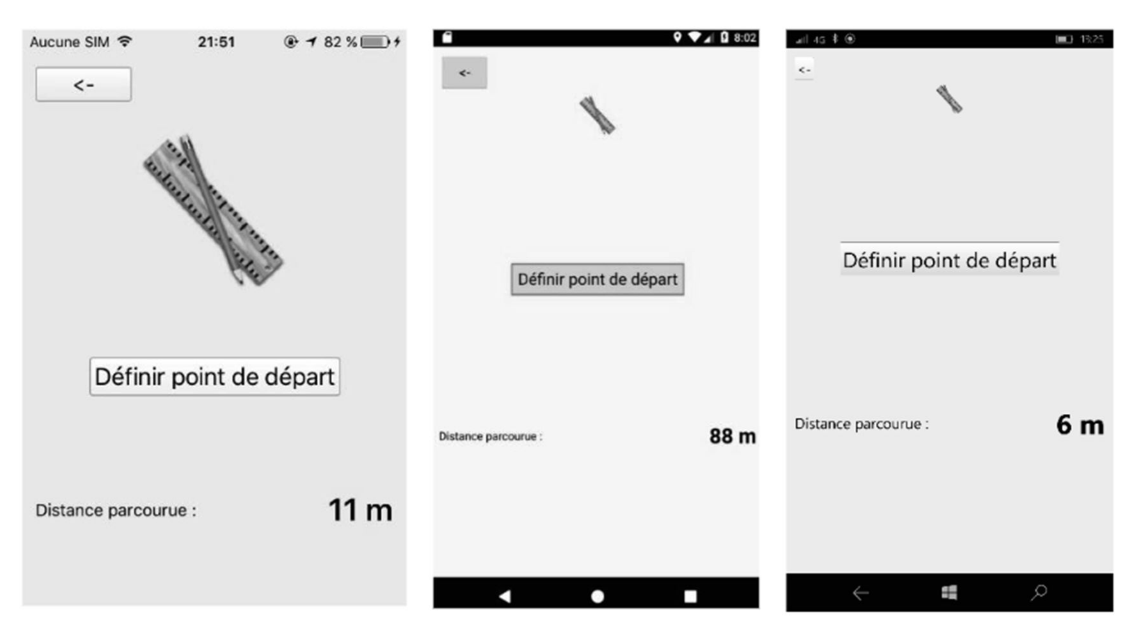
```
void Distance::nouvellePosition(const QGeoPositionInfo &info)
{
    courante = info.coordinate();
    afficheDistance();
}
```

L'affichage de la distance s'effectue simplement :

```
void Distance::afficheDistance()
{
    double dist = 0;
    if(base.isValid())
    {
        dist = courante.distanceTo(base);
    }
    ui->label_distance->setText(QString::number(dist,'f',0)+"
     m");
}
```

Le clic pour définir le point de départ se contente de stocker la position courante dans la position de base :

```
void Distance::on_pushButton_2_clicked() {
    base = courante;
}
```



Capture 83 : mesure de la distance avec Qt pour iOS, Android, Windows mobile

i) Améliorations possibles

Cette application est volontairement très simple et limitée pour pouvoir être décrite totalement en peu de pages… La première version que j'ai déposée sur les différents magasins d'applications est celle décrite dans ces pages. Il est possible que je réalise des mises à jour de l'application, vous pouvez quant à vous essayer de l'améliorer en ajoutant des fonctionnalités, en modifiant l'IHM, etc…

Voici quelques idées de modifications intéressantes :

- Améliorer l'IHM pour la rendre plus attrayante et « *responsive* »
- Généraliser l'utilisation des capteurs (luminosité, son, etc.)

5. Notes

Difficulté : facile

Plateformes ciblées : Windows 10, Android, iOS

Points techniques abordés : IHM, persistance des données

Langages utilisés : C#

Outils utilisés : Visual Studio, *Xamarin.Forms*

Téléchargement de l'application :

- Pour Android : https://frama.link/6_hnaKJH
- Pour Windows 10 : https://frama.link/sp8F7nLS
- Pour iOS : https://frama.link/gGB668Yk

Téléchargement des codes sources : https://frama.link/onsfWvEr

a) Description de l'application souhaitée

Un téléphone étant toujours à portée de main, il est intéressant de l'utiliser comme bloc-notes portable ! Nous allons donc écrire une petite application basique qui permettra de faire des petits pense-bêtes, sortes de « *post-it®* » virtuels.

b) Conception de l'application : design

L'application sera composée de deux écrans. Le premier écran regroupera l'ensemble des notes, le deuxième représentera le détail d'une note. On parle dans ce cas d'une application de type « maitre détail ».

Le premier écran présente l'ensemble des notes existantes, en indiquant leur titre et une couleur. Un bouton permet de créer une nouvelle note. Toucher une note permet de l'éditer. L'édition d'une note comme sa création vont utiliser le 2e écran.



Le 2e écran permet d'éditer le détail d'une note : son titre, un texte sur plusieurs lignes et une couleur de fond. Des couleurs pastel sont choisies pour faciliter la lecture de la note. Un bouton permet de revenir à l'écran maître (en sauvegardant les données), un autre permet de supprimer la note et de revenir également à l'écran maître.

c) Conception de l'application : couche métier

L'application manipule une notion métier : la note. Nous allons donc avoir une classe pour cette notion, et pour cela, suivre un modèle classique : la classe `Note` est le modèle (les données à stocker), l'interface

`SauveurNotes` est chargée du stockage, et `ControleurNotes` est le contrôleur, chef d'orchestre de la couche métier.

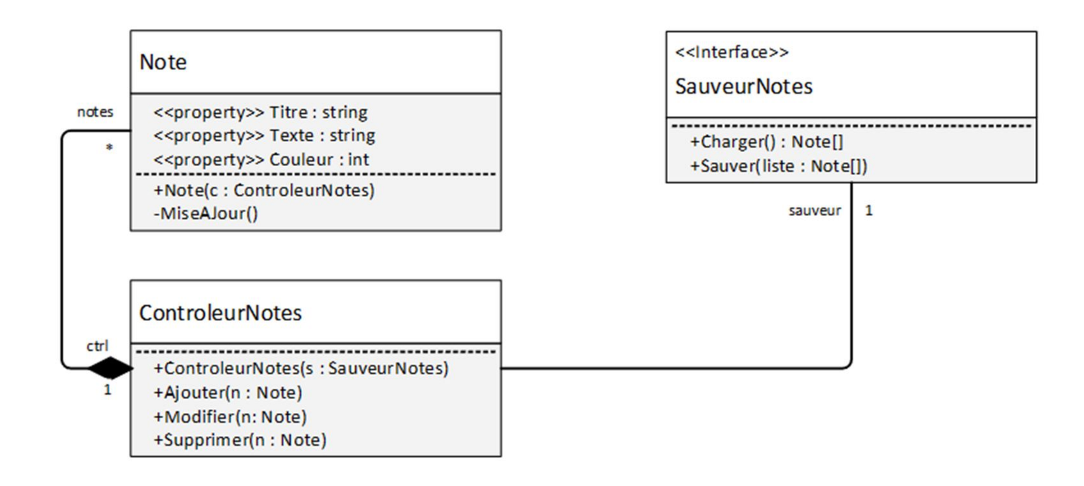


Diagramme 15 : couche métier de l'application Notes

d) Conception de l'application : couche IHM

La partie IHM étant composée de deux écrans, elle utilisera donc deux classes. Chaque classe sera associée au contrôleur :

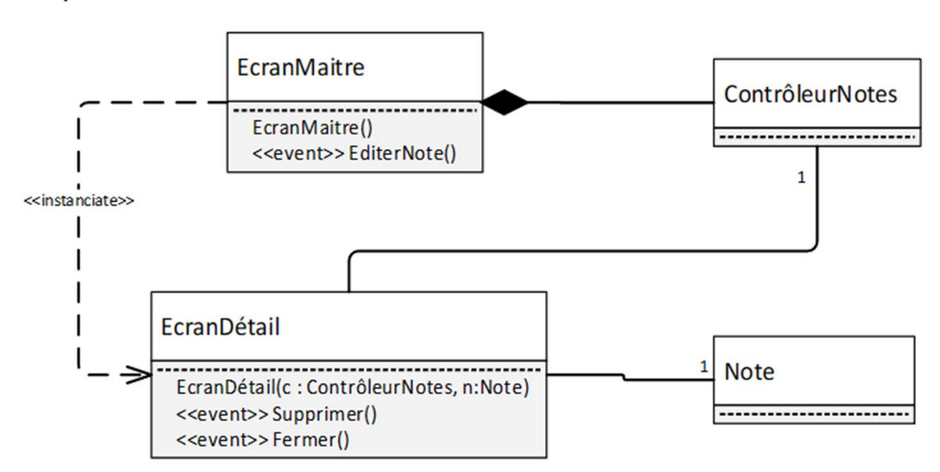


Diagramme 16 : couche IHM de l'application Notes

e) Développement de l'application : partie métier

Nous allons commencer par créer le projet pour l'application, en utilisant Visual Studio : créer une nouvelle application par le menu Visual C# → *Cross-platform*→ Application mobile (*Xamarin.Forms*) → Application Vide.

La classe `Note` est créée dans le projet .NET Standard, c'est presque un simple PODO[1], le seul comportement étant de prévenir le contrôleur lors d'une modification du modèle :

```
class Note
{
     private string titre;
     private string texte;
     private Color couleur;
     private ControleurNotes ctrl;

     public Note(ControleurNotes c=null)  {
        ctrl = c;
        titre = "";
        texte = "";
        couleur = Color.White;
     }
     public string Titre { get => titre; set {
        titre = value;
        if(ctrl!=null)
           ctrl.Modifier(this);
        }
     }
     public string Texte
     { get => texte;
        set
        {
           texte = value;
           if(ctrl!=null)
              ctrl.Modifier(this);
        }
     }
     public Color Couleur
     {
        get => couleur;
        set
        {
           couleur = value;
           if(ctrl!=null)
              ctrl.Modifier(this);
        }
     }
```

[1] PODO : *plain old data object* : « bon vieil objet de données », objet simple ne comportant que des données et pas de comportement. Parfois appelé PODS (*plain old data structure*).

```
    public void Connecte(ControleurNotes ctrl)
    {
      this.ctrl = ctrl;
    }
}
```

L’interface pour la couche de persistance est simple :

```
    interface SauveurNotes
    {
        Note[] Charger();
        void Sauver(Note[] notes);
    }
```

Le contrôleur est associé à un « sauveur » ainsi qu’au modèle (les notes) :

```
private List<Note> notes = new List<Note>();
private SauveurNotes sauveur;
private List<AfficheurNotes> afficheurs = new
     List<AfficheurNotes>();
```

Il contient des opérations pour ajouter ou supprimer une note, et mettre à jour le stockage :

```
private void ListeChangee()
{
     sauveur.Sauver(notes.ToArray());
}
public void Ajouter(Note n)
{
     notes.Add(n);
     ListeChangee();
}
public void Supprimer(Note n) {
     notes.Remove(n);
     ListeChangee();
}
```

Il ne reste plus qu’à ajouter le constructeur (qui initialise le modèle également) ainsi qu’un accesseur en lecture sur le modèle :

```
public ControleurNotes(SauveurNotes sauveur)
{
     this.sauveur = sauveur;
     notes.AddRange(sauveur.Charger());
     foreach (Note n in notes)
       n.Connecte(this);
```

```
}
public Note[] Notes
{
    get => notes.ToArray();
}
```

Une classe pour tester sans avoir à gérer la persistance est facile à faire :

```
class FakeSauveur : SauveurNotes
{
    public Note[] Charger()
    {
      Note[] notes = new Note[2];
      notes[0] = new Note() { Titre = "Courses",
        Texte = "Acheter de la bière, des chips et du
    pain",
        Couleur = Color.LightBlue };
      notes[1] = new Note() { Titre = "A faire",
        Texte = "Régler la PI3 pour Retropie",
        Couleur = Color.Wheat };
      return notes;
    }
    public void Sauver(Note[] notes) {} // rien à faire
}
```

f) Développement de l'application : écran « maître »

Cet écran va contenir un `Label` pour le titre, un `Button` pour ajouter une note, et une `ListView` pour afficher les différentes notes. La partie visuelle est donc à décrire en XAML. Nous allons utiliser une disposition verticale (`StackLayout`), avec une disposition horizontale pour le haut de la page :

```
<StackLayout Orientation="Vertical" Padding="10" >
    <StackLayout Orientation="Horizontal" Padding="10"
    BackgroundColor="LightBlue" >
      <Label Text="Mes notes" FontSize="Large"
    FontAttributes="Bold" HorizontalOptions="StartAndExpand"
    VerticalOptions="Center"/>
      <Button Image="add.png" BackgroundColor="LightBlue"
    HorizontalOptions="End" VerticalOptions="Fill"/>
    </StackLayout>
```

Vient ensuite la définition de la liste, en utilisant un modèle de liste et le *data binding* avec le modèle (la note) :

```
    <ListView x:Name="notes" Margin="10"
    HorizontalOptions="FillAndExpand"
    VerticalOptions="FillAndExpand">
      <ListView.ItemTemplate>
        <DataTemplate>
          <ViewCell Tapped="Detail">
            <Grid BackgroundColor="{Binding Couleur}"
    Margin="10" Padding="10">
              <Label Text="{Binding Titre}"
    HorizontalOptions="StartAndExpand" FontSize="Large"  />
            </Grid>
          </ViewCell>
        </DataTemplate>
      </ListView.ItemTemplate>
    </ListView>
</StackLayout>
```

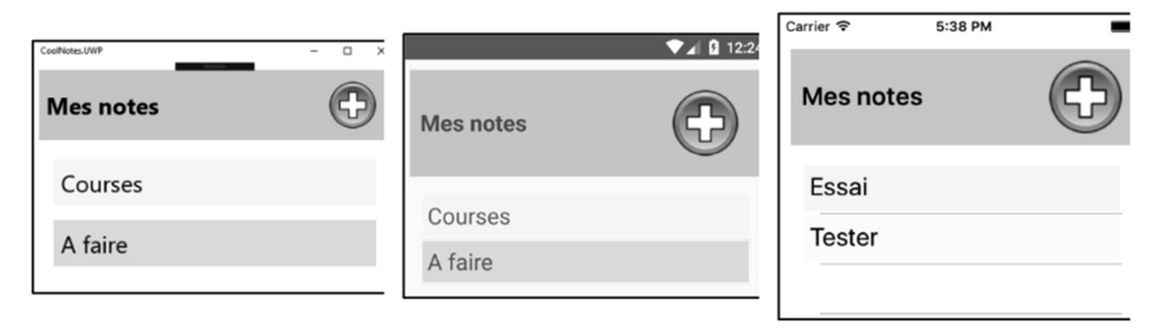
Dans la partie code (C#) de cette page, commençons par les associations de la page (donc ses attributs) :

```
private ControleurNotes controleur;
private SauveurNotes sauveur;
```

Dans un premier temps nous utiliserons la fausse couche de persistance. Dans le constructeur de la page, il faut initialiser les associations pour inscrire la page au contrôleur, puis appeler la modification de liste pour mettre à jour l'affichage :

```
public MainPage()
{
    InitializeComponent();
    sauveur = new FakeSauveur();
    controleur = new ControleurNotes(sauveur);
    controleur.Inscrire(this);
    ModificationListe();
}
```

Cette partie peut d'ores et déjà être testée :



Capture 84 : écran maître de l'application Notes, sous Windows, Android et iOS

g) Développement : page « détails »

Le 2e écran présente la vue détaillée d'une note. Il est utilisé à la fois pour créer et pour modifier une note existante. Commençons par définir la vue XAML de la page, basée sur une disposition verticale et des dispositions horizontales.

```
<StackLayout Orientation="Vertical" Padding="10" >
```

La première ligne de l'écran contiendra le titre et deux boutons :

```
<StackLayout VerticalOptions="Start" Orientation="Horizontal"
     Padding="10" BackgroundColor="LightBlue">
     <Button Image="back.png" BackgroundColor="LightBlue"
     HorizontalOptions="Start" VerticalOptions="Fill"
     Clicked="Retour"/>
     <Entry Text="{Binding Titre}" FontSize="Large"
     HorizontalTextAlignment="Center"
     HorizontalOptions="FillAndExpand" />
     <Button Image="remove.png" BackgroundColor="LightBlue"
     HorizontalOptions="End" VerticalOptions="Fill"
     Clicked="Supprimer" />
</StackLayout>
```

XAML

Au centre, une zone de saisie multiligne, dont le fond est de la couleur de la note et qui contient le texte de la note :

```
<Editor x:Name="saisie"  Text="{Binding Texte}"
     BackgroundColor="{Binding Couleur}"
     HorizontalOptions="Fill" VerticalOptions="FillAndExpand"
     />
```

Enfin, en bas de page, des boutons de couleur pour la couleur de fond (j'ai choisi des teintes douces mais vous pouvez en changer) :

```
<StackLayout VerticalOptions="End" Orientation="Horizontal"
     Padding="4">
     <Button BackgroundColor="Wheat" BorderColor="Black"
     HorizontalOptions="FillAndExpand" HeightRequest="60"
     Clicked="Changer" />
     <Button BackgroundColor="Azure" BorderColor="Black"
     HorizontalOptions="FillAndExpand" HeightRequest="60"
     Clicked="Changer"/>
     <Button BackgroundColor="Ivory" BorderColor="Black"
     HorizontalOptions="FillAndExpand" HeightRequest="60"
     Clicked="Changer"/>
     <Button BackgroundColor="Lavender" BorderColor="Black"
     HorizontalOptions="FillAndExpand" HeightRequest="60"
     Clicked="Changer"/>
     <Button BackgroundColor="LightYellow"
     BorderColor="Black" HorizontalOptions="FillAndExpand"
     HeightRequest="60" Clicked="Changer"/>
     <Button BackgroundColor="PeachPuff" BorderColor="Black"
     HorizontalOptions="FillAndExpand"  HeightRequest="60"
     Clicked="Changer"/>
</StackLayout>
```

Passons à la partie code C# de cette page.

La page est associée à un contrôleur, et reçoit également une note en paramètre : celle-ci deviendra notre « *binding context* », l'objet « lié » aux contrôles :

```
private ControleurNotes controleur;
public DetailNote (ControleurNotes ctrl, Note note) {
     InitializeComponent ();  BindingContext = note;
     controleur = ctrl; }
```

Le clic sur le bouton de retour provoque le retour à la page précédente :

```
private void Retour(object sender, EventArgs args)
{
     Navigation.PopModalAsync();
}
```

Le clic sur le bouton de suppression provoque la suppression de la note et le retour à la page précédente :

```
private void Supprimer(object sender, EventArgs args)
{
     if(BindingContext is Note n)
     {
```

```
        controleur.Supprimer(n);
        Navigation.PopModalAsync();
    }
}
```

Le clic sur l'un des boutons de couleur provoque le changement de la couleur de la note et du fond de la zone de saisie :

```
private void Changer(object sender, EventArgs args)
{
    if(sender is Button b)
    {
        Color c = b.BackgroundColor;
        if(BindingContext is Note n)
        {
            n.Couleur = c;
            saisie.BackgroundColor = c;
        }
    }
}
```

Repassons ensuite à la page principale pour « appeler » la page de détail.

Le clic sur le bouton « ajouter » déclenche la création d'une nouvelle note et le passage à la 2e page :

```
private void Ajouter(object sender, EventArgs args) {
    Note n = new Note(controleur);
    controleur.Ajouter(n);
    DetailNote details = new DetailNote(controleur, n);
    Navigation.PushModalAsync(details);
}
```

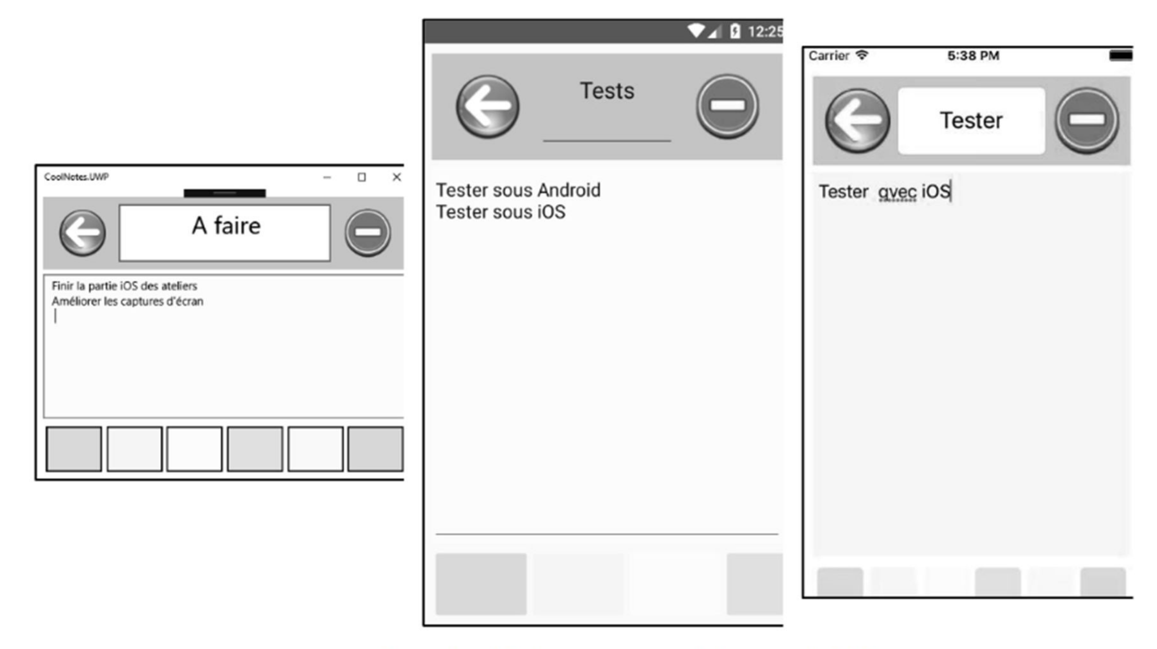
Le clic sur un élément de la liste entraine un accès à la 2ème page :

```
private void Detail(object sender, EventArgs args)
{
    if(notes.SelectedItem is Note n)
    {
        DetailNote details = new DetailNote(controleur,n);
        Navigation.PushModalAsync(details);
    }
}
```

Il ne reste plus qu'à redéfinir l'évènement appelé quand la page apparaît (déclenché lors du retour de la 2e page) pour garantir une mise à jour de l'affichage :

```
protected override void OnAppearing()
{
     base.OnAppearing();
     ModificationListe();
}
```

Le 2ème écran peut alors être testé :



Capture 85 : écran de saisie de notes, sous Windows, Android, iOS

h) Développement : persistance

Pour le stockage persistant des données, nous allons recourir au procédé le plus simple : les paramètres. La limitation en taille ne devrait pas poser de problème, sauf en cas de très nombreuses notes (auquel cas utilisez plutôt des fichiers).

Nous pouvons soit utiliser un *plugin* pour la gestion multiplateforme des paramètres, soit écrire nous-mêmes le code pour chaque plateforme, dans chaque projet. Le plus simple est bien entendu d'utiliser le *plugin*… Installez donc le *plugin NuGet Xam.Plugins.Settings* (voir VII.1.d) dans tous les projets de votre solution.

Nous allons utiliser la sérialisation JSON pour créer une chaîne compacte à partir de nos notes. Il faut donc modifier notre classe `Note` pour qu'elle soit sérialisable : marquer la classe comme `[DataContract]` et les attributs comme `[DataMember]`.

Il faut ensuite créer une classe qui implémente `SauveurNotes` pour les sauver en JSON dans les paramètres :

```
class SauveurParams : SauveurNotes
{
     public Note[] Charger()
     {
       Note[] notes;
       string s =
     CrossSettings.Current.GetValueOrDefault("notes", "");
       if (s == "")
       {
         notes = new Note[0];
       }
       else
       {
         try
         {
           using (MemoryStream flux = new
     MemoryStream(Encoding.UTF8.GetBytes(s)))
           {
             DataContractJsonSerializer ser = new
     (typeof(Note[]));
             notes = ser.ReadObject(flux) as Note[];
           }
         }
         catch
         {
           notes = new Note[0];
         }
       }
       return notes;
     }

     public void Sauver(Note[] notes)      {
       using (MemoryStream flux = new MemoryStream())
       {
         DataContractJsonSerializer ser = new
     DataContractJsonSerializer(typeof(Note[]));
         ser.WriteObject(flux, notes);
           string s =
     Encoding.UTF8.GetString(flux.ToArray());
           CrossSettings.Current.AddOrUpdateValue("notes",
     s);
         }
       }
     }
}
```

i) Améliorations possibles

Cette application est volontairement très simple et limitée pour pouvoir être décrite totalement en peu de pages… La première version que j'ai déposée sur les différents magasins d'applications est celle décrite dans ces pages. Il est possible que je réalise des mises à jour de l'application, vous pouvez quant à vous essayer de l'améliorer en ajoutant des fonctionnalités, en modifiant l'IHM, etc…

Voici quelques idées de modifications intéressantes :

- Permettre de poser une note sur le « bureau » du téléphone : *widget* pour Android, tuile pour Windows 10…
- Intégrer plus d'informations dans la note (date par exemple)
- Permettre de rechercher une note
- Hiérarchiser les notes avec des dossiers, sous-dossiers…
- Classifier les notes par catégorie…

6. Compteur de points au Tarot

Difficulté : moyen

Plateformes ciblées : Windows 10, Android, iOS

Points techniques abordés : IHM multi-écrans, persistance des données

Langages utilisés : C#

EDI utilisés : Visual Studio, *Xamarin.Forms*

Téléchargement de l'application :

- Pour Android : https://frama.link/-Sc08Vvh
- Pour Windows : https://frama.link/EKgCyZn-
- Pour iOS : https://frama.link/Rz1KG0A1

Téléchargement des codes sources : https://frama.link/kUFd7mTv

a) Descriptif de l'application souhaitée

Le but de cette application est de faciliter le comptage des points du jeu de tarot (voir http://fftarot.fr/ , le site de la Fédération Française de Tarot pour un détail des différentes règles).

Nous retiendrons les règles officielles pour le jeu à 3, 4 ou 5 joueurs. Les cas du chelem, de la « misère », ainsi que du jeu à 6 (avec un mort) ne sont pas abordés.

L'application devra permettre de saisir les informations de chaque partie afin de comptabiliser et stocker automatiquement les scores. Bien sûr, un jeu pouvant se dérouler sur une période de temps importante, il sera nécessaire de gérer la persistance des données ; à ce titre, cet atelier est une application directe du chapitre VII. L'application contiendra également une interface utilisateur importante, avec plusieurs écrans, et donc sera en outre une application du chapitre VI.5.b).

Une **partie** de tarot est composée de plusieurs **tours** et occupe 4 (3, 5…) **joueurs**. A chaque **tour**, un **joueur** est le **preneur**, et choisit un **contrat** (petite, garde, etc…). A la fin du tour, on compte les **points**. Suivant le nombre ***d'oudlers*** (les « bouts ») obtenus par le preneur à la fin, il doit faire un certain **score**. On indique le score réellement obtenu, le programme calcule la différence ainsi que les points pour chaque joueur. L'application va utiliser des ressources textuelles et visuelles (voir chapitre V) et va devoir sauvegarder des données (voir chapitre VII).

b) Conception de l'application : *design*

L'application devra être la plus simple à utiliser possible et présenter une ergonomie efficace.

Le téléphone sera utilisé en mode « portrait » ce qui est plus simple à tenir à une seule main.

Le premier écran (accueil) permet à l'utilisateur de commencer une nouvelle partie ou de reprendre une partie en cours.



Capture 87 : maquette écran d'accueil application tarot

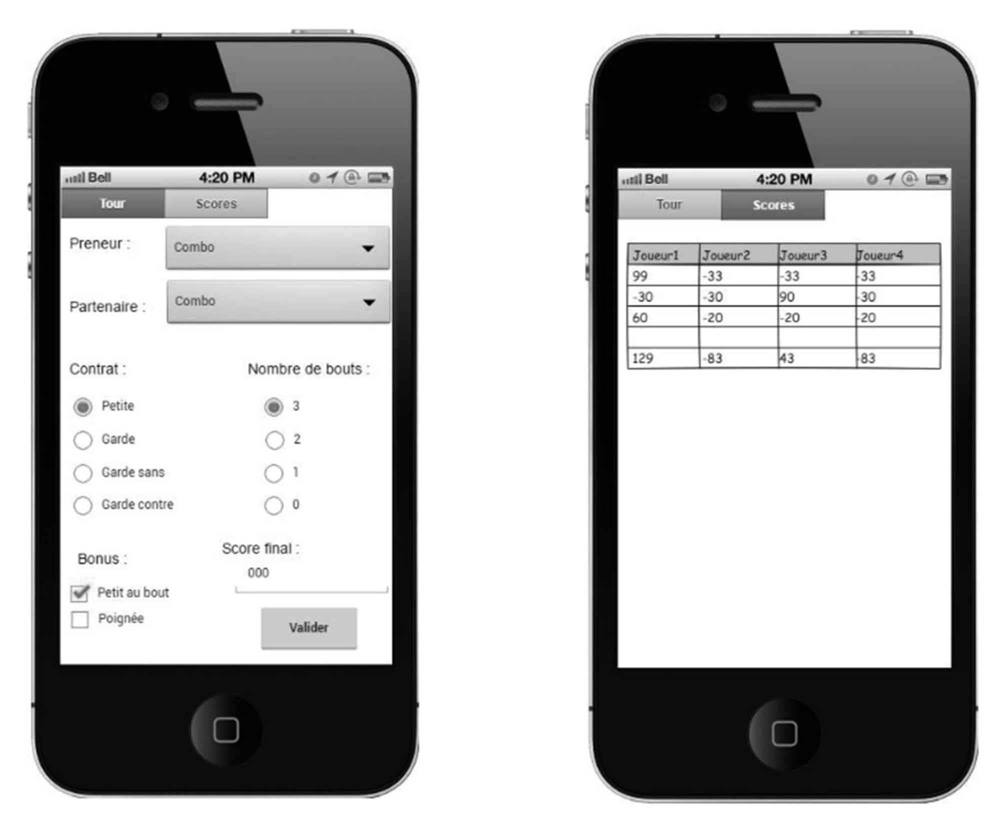
L'écran suivant sert à configurer une partie en entrant le nom des joueurs (cas d'une nouvelle partie). Puisqu'il peut y avoir plusieurs joueurs, on permet à l'utilisateur d'entrer entre 3 et 5 noms.



Capture 86 : maquette écran nouvelle partie tarot

L'écran suivant sert pour une partie. Il est découpé en deux onglets, l'un pour configurer le tour de jeu, l'autre qui liste les scores.

La configuration du tour de jeu permet de choisir le preneur, son partenaire (dans le cas du jeu à 5), son contrat, son nombre *d'oudlers* (bouts) et son score final.



Capture 88 : maquette écran partie tarot

c) Conception : architecture générale

Nous allons découper notre application en trois parties :

- La gestion de l'interface utilisateur (IHM)
- La gestion des données (métier)
- La gestion de la persistance (stockage)

Cette architecture peut se représenter par le diagramme de packages UML suivant :

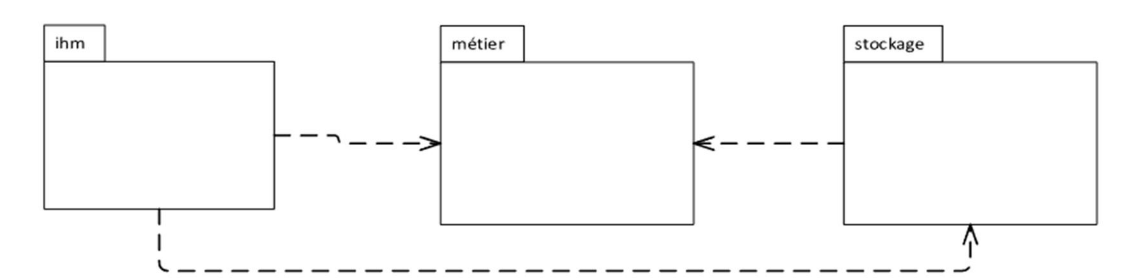


Diagramme 17 : architecture de l'application tarot

d) Conception : paquet « métier »

Cette partie comprend toutes les classes nécessaires pour représenter les entités manipulées par le jeu de tarot. Nous avons besoin de la notion de `Joueur`, de `Partie`, de `Tour`, de `preneur`, de `partenaire`…

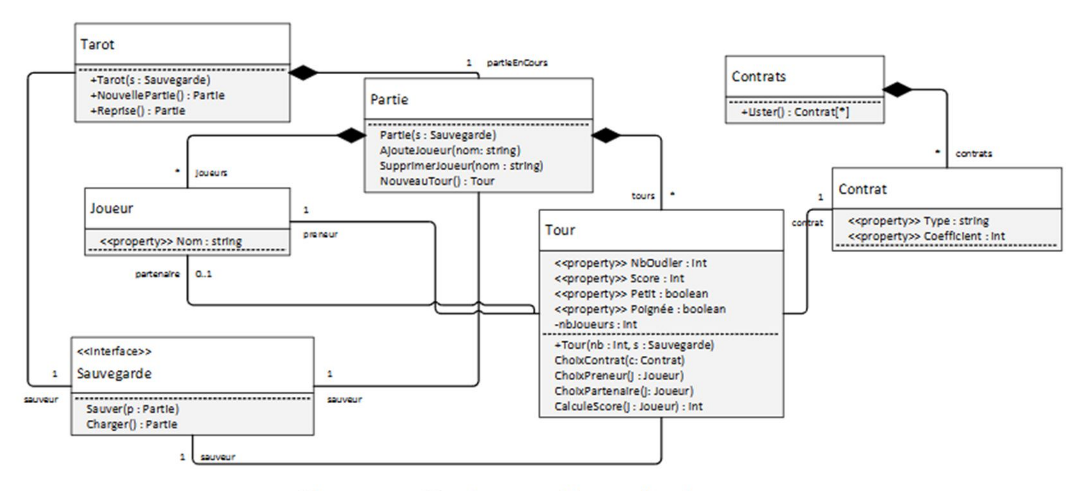


Diagramme 18 : classes métier application tarot

e) Conception : paquet « IHM »

Cette partie va comprendre les classes représentant les différents écrans ; celles-ci contiendront les gestionnaires d'évènements liés aux interactions utilisateur et des liens avec la partie « métier » et la partie « sauvegarde ».

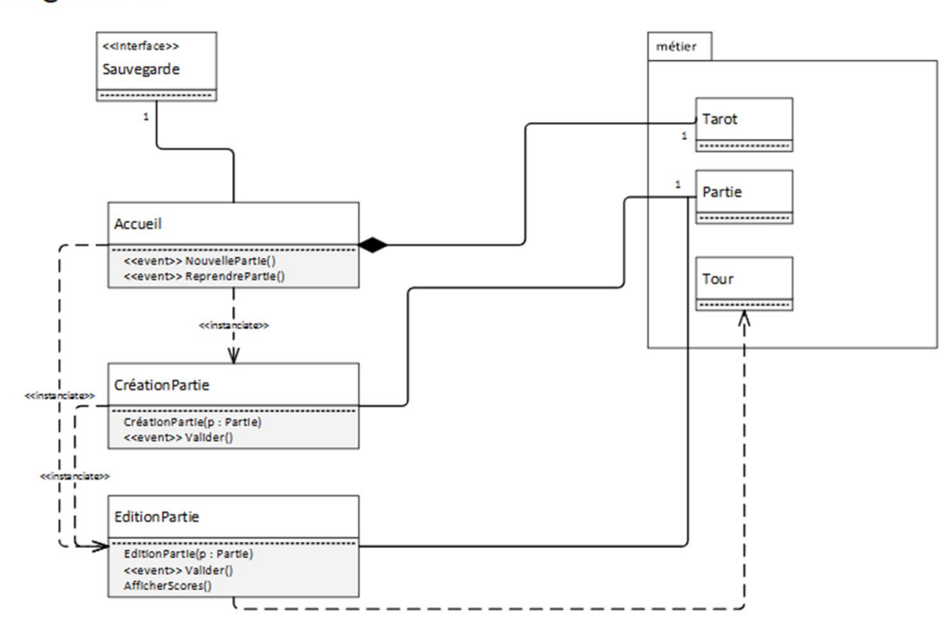


Diagramme 19: conception ihm application tarot

f) Conception : paquet « stockage »

Ce paquet ne contiendra qu'une seule classe, implémentation de l'interface « `Sauvegarde` ». Son implémentation dépendra fortement de la technologie retenue.

g) Développement : page d'accueil

Chaque écran décrit dans la conception sera implémenté par une page *xaml*. Nous avons donc 3 pages dont une est de type « à onglets ». Nous utiliserons des ressources localisées (une en français au moins) pour les différentes chaînes afin de faciliter l'internationalisation de l'application. Nous allons développer écran par écran, avec l'ensemble des fonctionnalités.

Pour avoir des ressources, il faut créer dans le projet partagé un dossier `Textes` contenant une ressource par langue voulue (ici `Strings.resx` et `Strings.en.resx`) ainsi qu'un dossier `Images` qui contiendra un fichier image affiché dans l'écran.

Images
tarot.png
Textes
Strings.en.resx
Strings.resx

Commençons par la vue : un composant `Image`, un `label`, deux boutons, le tout dans un `StackLayout`. Le code XAML donne :

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
     xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
xmlns:local="clr-namespace:TarotCompteur"
xmlns:resx="clr-namespace:TarotCompteur.Textes"
Title="{x:Static resx:Strings.Titre}"
x:Class="TarotCompteur.MainPage">
<ContentPage.Content> <StackLayout>
       <Image x:Name="image" HorizontalOptions="Center"
     VerticalOptions="Start" />
       <Label Text="{x:Static resx:Strings.Titre}"
     FontAttributes="Bold" FontSize="Large"
     HorizontalOptions="Center"/>
       <Button Text="{x:Static resx:Strings.NouvPartie}"
     Margin="10" HorizontalOptions="Center"
     Clicked="NouvellePartie"/>
       <Button Text="{x:Static resx:Strings.Reprise}"
     Margin="10" HorizontalOptions="Center"
     Clicked="Reprise"/> </StackLayout>
</ContentPage.Content> </ContentPage>
```

XAML

Dans le code associé, nous avons besoin de rajouter dans le constructeur le chargement de l'image à partir des ressources :

```
public MainPage() {
     InitializeComponent();
     var assembly = typeof(MainPage).GetTypeInfo().Assemby ;
      image.Source =
     ImageSource.FromResource("TarotCompteur.Images.tarot.png
     ", assembly);
}
```

Le test de ce premier écran donne le résultat suivant :







Capture 89: écran d'accueil de l'application tarot (windows, android, ios)

Il faut ensuite aller dans le *package* métier pour créer les classes `Tarot`, `Partie` et l'interface `Sauvegarde`.

```
interface ISauvegarde {
     void Sauver(Partie p);
     Partie Charger();
}
```

La classe `Partie` ne contient pour l'instant que l'association avec le stockage :

```
class Partie {
     private ISauvegarde stockage;
     public ISauvegarde Stockage {get=>stockage ;
     set=>stockage=value ;}
     public Partie(ISauvegarde s)   {
        stockage = s;
     }
}
```

La classe `Tarot` est assez simple pour l'instant :

```
class Tarot
{
     private ISauvegarde stockage;
     private Partie partieEnCours;

     public Tarot(ISauvegarde s)    {
        stockage = s;
     }

     public Partie NouvellePartie()
     {
        partieEnCours = new Partie();
        return partieEnCours;
     }

     public Partie Reprise()
     {
        partieEnCours = stockage.Charger();
        return partieEnCours;
     }
}
```

Il ne reste qu'à lier les couches, en ajoutant dans la classe de la page d'accueil les attributs correspondants aux associations :

```
private ISauvegarde stockage;
private Tarot jeu;
```

Pour le test nous allons créer une sauvegarde factice (qui ne fait rien) :

```
class SauvegardeFactice : ISauvegarde
{
     public Partie Charger()
     {
        return new Partie(this);
     }

     public void Sauver(Partie p) {
     }
}
```

Il ne reste qu'à ajouter dans le constructeur de l'écran d'accueil l'initialisation des liens :

```
stockage = new SauvegardeFactice();
jeu = new Tarot(stockage);
```

Les opérations de réponse aux évènements ne font que « lancer » la partie pour l'instant :

```
private void NouvellePartie(object sender, EventArgs e)
{
     Partie p = jeu.NouvellePartie();
}

private void Reprise(object sender, EventArgs e)
{
     Partie p = jeu.Reprise();
}
```

L'application comportant plusieurs pages, il faut que la page principale soit de type « `NavigationPage` », et donc modifier comme vu au V.5, le fichier principal de l'application (`App.xaml.cs`) de la façon suivante :

```
MainPage = new NavigationPage(new MainPage());
```

Le travail sur la première page étant terminé, nous allons passer à la page suivante.

h) Développement : page de création de partie

Cette page est une simple page XAML. Nous utiliserons une disposition de type empilement vertical (`StackLayout`), des étiquettes (`Label`), zones de saisie (`Entry`), boutons (`Button`) et liste (`ListView`) (comme pour la page précédente, les textes seront issus des fichiers ressources pour faciliter la traduction) :

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
     xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
xmlns:resx="clr-namespace:TarotCompteur.Textes"
x:Class="TarotCompteur.PageCreerPartie">

<ContentPage.Content>

<StackLayout Orientation="Vertical">
```

```
    <StackLayout Orientation="Horizontal">
      <Label Text="{x:Static resx:Strings.Joueur}"
    Margin="10" HorizontalTextAlignment="End" />
      <Entry x:Name="nom" Margin="10"
    HorizontalTextAlignment="Center"
    HorizontalOptions="FillAndExpand"/>
    </StackLayout>
    <StackLayout Orientation="Horizontal"
    HorizontalOptions="CenterAndExpand">
      <Button Text="{x:Static resx:Strings.Ajouter}"
    Margin="10" Clicked="Ajouter" />
      <Button Text="{x:Static resx:Strings.Supprimer}"
    Margin="10" Clicked="Supprimer"/>
    </StackLayout>
    <Label Text="{x:Static resx:Strings.Joueurs}"
    Margin="10" HorizontalOptions="Start"/>
    <ListView x:Name="joueurs" Margin="10"
    HorizontalOptions="Fill" VerticalOptions="Fill"
    BackgroundColor="Beige"/>
    <Button x:Name="debut" IsEnabled="False" Text="{x:Static
    resx:Strings.Debut}" Margin="30,10,30,10"
    VerticalOptions="End" Clicked="Commencer"/>
</StackLayout>
</ContentPage.Content>
</ContentPage>
```

Il faut modifier le premier écran pour permettre la navigation jusqu'à cette page :

```
private async void NouvellePartie(object sender, EventArgs e)
{
    Partie p = jeu.NouvellePartie();
    Page page = new PageCreerPartie(p);
    await Navigation.PushAsync(page);
}
```

Lorsque l'utilisateur clique sur le bouton « Ajouter », il doit intégrer un joueur à la partie. Commençons par créer la classe `Joueur` :

```
public class Joueur
{
    private string nom;

    public string Nom {
      get => nom;
      set
      {
```

```
            if (value == null || value == "")
               throw new Exception(Textes.Strings.NomVide);
            nom = value;
         }
    }

      public override string ToString()       {
         return nom;
      }

      public bool EstEgal(Joueur j)
      {
         bool ok = false;
         if (j != null && j.nom == nom)
            ok = true;
         return ok;
      }
}
```

Il faut ensuite modifier la classe `Partie` pour qu'elle contienne un attribut de type tableau (ou liste) de joueurs, ainsi qu'une opération pour ajouter un joueur. Une partie de tarot ne pouvant pas avoir plus de 5 joueurs, cette opération lèvera une exception si l'on tente d'en ajouter trop. Le nom du joueur devant être unique dans la partie, une exception sera levée s'il existe déjà.

```
public void Ajouter(Joueur j)
{
      if (joueurs.Count == 5)
         throw new Exception(Textes.Strings.PartieComplete);
      if (joueurs.Find((x) => x.Nom == j.Nom) != null)
         throw new Exception(Textes.Strings.NomExiste);
      joueurs.Add(j);
}
```

La liste des joueurs est obtenue simplement :

```
public Joueur[] Lister() {
      return joueurs.ToArray();
}
```

Dans l'opération de réponse au clic du bouton (classe de la page de création de partie), il suffit d'appeler cette opération (en prenant bien soin de capturer les exceptions pour afficher un message). Nous allons également

mettre à jour l'état du bouton pour commencer la partie, il ne sera actif que si le nombre de joueurs est correct (entre 3 et 5) :

```
private void Ajouter(object sender, EventArgs e)
{
     try
     {
       Joueur j = new Joueur() { Nom = nom.Text };
       partie.Ajouter(j);
       nom.Text = "";
       MajJoueurs();
     }
     catch(Exception x)
     {
       DisplayAlert(Textes.Strings.Erreur, x.Message, "OK");
     }
}
private void MajJoueurs()
{
     Joueur[] tab = partie.Lister();
     joueurs.ItemsSource = tab;
     debut.IsEnabled = tab.Length >= 3;
}
```

Pour la suppression, il faut récupérer le joueur sélectionné depuis la liste, l'enlever de la partie (il faut donc une fonction *ad hoc* dans la classe `Partie`) puis rafraîchir l'affichage :

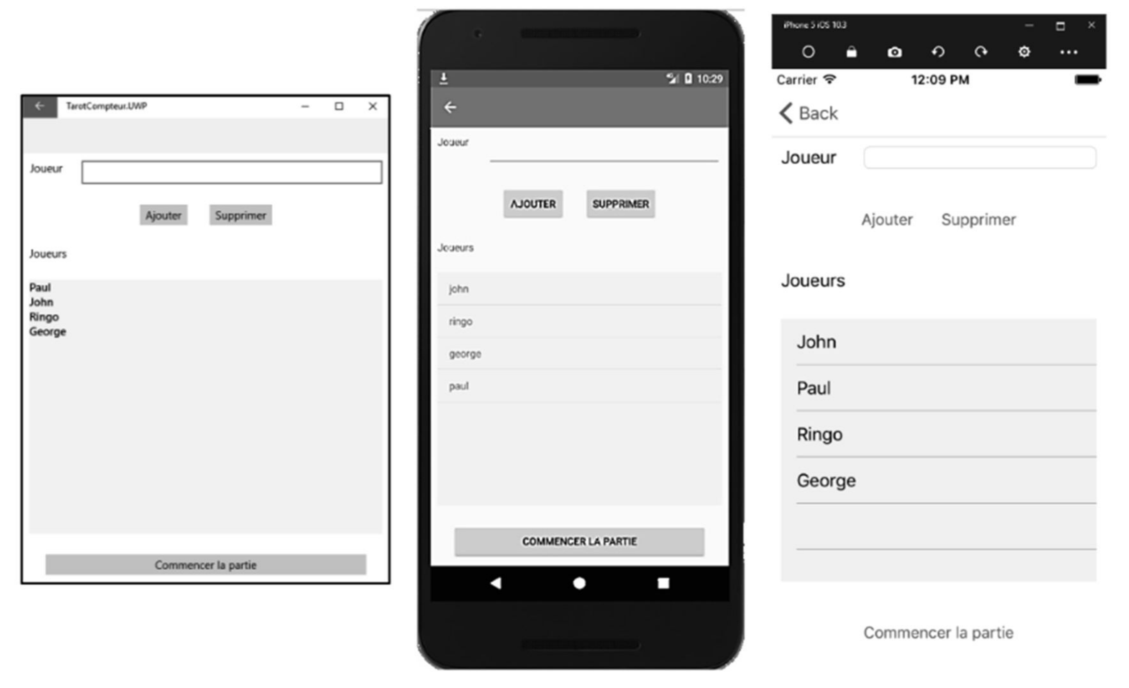
```
private void Supprimer(object sender, EventArgs e)
{
     Joueur j = joueurs.SelectedItem as Joueur;
     partie.Supprimer(j);
     MajJoueurs();
}
```

La fonction suivante est donc nécessaire dans `Partie` :

```
public void Supprimer(Joueur j)
{
     joueurs.Remove(j);
}
```

Le test est à présent possible : il faudra avoir entre 3 et 5 joueurs pour pouvoir cliquer sur le bouton en bas de l'écran.

Le clic sur le bouton « Commencer la partie » va donc faire passer à l'écran suivant (la partie).



Capture 90 : écran création de partie de tarot (windows, android, iOS)

i) Développement : page de partie

Cette page est une « page à onglets » (type `TabbedPage`), contenant deux onglets.

Le début de la page est le suivant :

```
<TabbedPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
xmlns:resx ="clr-namespace:TarotCompteur.Textes"
    x:Class="TarotCompteur.PagePartie">
```

Vient ensuite la disposition des onglets.

Le premier onglet (le tour de jeu) contient des étiquettes (`Label`), des listes déroulantes (`Picker`), des contrôles oui/non (`Switch`). Il n'existe pas d'équivalent au bouton radio en *Xamarin.Forms*, nous utiliserons donc des listes déroulantes ou des zones de saisie.

La disposition sera de type grille (`Grid`).

```
<ContentPage Title="{x:Static resx:Strings.Tour}" >
<Grid>
<Grid.RowDefinitions>
    <RowDefinition Height="*"/>
```

```
    <RowDefinition Height="*"/>
    <RowDefinition Height="*"/>
    <RowDefinition Height="*"/>
    <RowDefinition Height="*"/>
    <RowDefinition Height="*"/>
    <RowDefinition Height="*"/>
    <RowDefinition Height="*"/>
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="*"/>
</Grid.ColumnDefinitions>
<Label Text="{x:Static resx:Strings.Preneur}" Grid.Row="0"
    Grid.Column="0" HorizontalOptions="End"
    VerticalOptions="Center" />
<Picker x:Name="preneur" Grid.Row="0" Grid.Column="1"
    Margin="6"/>
<Label Text="{x:Static resx:Strings.Partenaire}" Grid.Row="1"
    Grid.Column="0" HorizontalOptions="End"
    VerticalOptions="Center"/>
<Picker x:Name="partenaire" IsEnabled="False" Grid.Row="1"
    Grid.Column="1" />
<Label Text="{x:Static resx:Strings.Contrat}" Grid.Row="2"
    Grid.Column="0" HorizontalOptions="Center"
    VerticalOptions="End" />
<Label Text="{x:Static resx:Strings.Oudlers}" Grid.Row="2"
    Grid.Column="1" HorizontalOptions="Center"
    VerticalOptions="End"/>
<Picker x:Name="contrats" Grid.Row="3" Grid.Column="0"
    Margin="6" />
<StackLayout Orientation="Horizontal" Grid.Row="3"
    Grid.Column="1" >
    <Slider x:Name="bouts" HorizontalOptions="FillAndExpand"
    Margin="4" Minimum="0" Maximum="3" Value="0"
    ValueChanged="NombreBouts"/>
    <Label x:Name="nbBouts" HorizontalOptions="End"
    Margin="4" Text="0" />
</StackLayout>
<Label Text="{x:Static resx:Strings.Petit}" Grid.Row="4"
    Grid.Column="0" HorizontalOptions="End"
    VerticalOptions="Center" />
<Switch x:Name="petit" Grid.Row="4" Grid.Column="1"
    VerticalOptions="Center" />
<Label Text="{x:Static resx:Strings.Poignee}" Grid.Row="5"
    Grid.Column="0" HorizontalOptions="End"
    VerticalOptions="Center"/>
<Switch x:Name="poignee" Grid.Row="5" Grid.Column="1"
    VerticalOptions="Center"/>
```

```
<Label Text="{x:Static resx:Strings.ScoreFinal}" Grid.Row="6"
     Grid.Column="0" HorizontalOptions="End"
     VerticalOptions="Center"/>
<Entry x:Name="score" Grid.Row="6" Grid.Column="1" Margin="6"
     Keyboard="Numeric"/>
<Button Text="{x:Static resx:Strings.Valider}" Grid.Row="7"
     Grid.ColumnSpan="2" HorizontalOptions="Center"
     Margin="10"/>
</Grid>
</ContentPage>
```

Dans le code de cette page, il faut tout d'abord ajouter un attribut du type `Partie` :

```
private Partie partie;
```

Cet attribut sera initialisé dans le constructeur de la page. Dans ce même constructeur, on va utiliser l'attribut pour initialiser les différents contrôles.

Nous avons déjà une opération `Partie.Lister` pour lister les joueurs. Il faut à présent créer la classe `Contrat` :

```
public class Contrat
{
     private int coefficient;
     private string type;

     public Contrat(string nom, int coef)
     {
        coefficient = coef;
        type = nom;
     }

     public int Coefficient { get => coefficient; }
     public string Type { get => type; }

     public override string ToString()
     {
        return type;
     }
}
```

Puis ajouter dans la partie un tableau reprenant les contrats possibles :

```
private Contrat[] contrats = new Contrat[4];
```

Tableau qui sera initialisé dans les constructeurs de `Partie` :

```
contrats[0] = new Contrat(Textes.Strings.Petite, 1);
contrats[1] = new Contrat(Textes.Strings.Garde, 2);
contrats[2] = new Contrat(Textes.Strings.GardeSans, 4);
contrats[3] = new Contrat(Textes.Strings.GardeContre, 6);
```

Une propriété en lecture seule est également utile :

```
public Contrat[] Contrats { get => contrats; }
```

Dans le constructeur de la classe de la page partie, il convient d'initialiser les listes déroulantes avec les noms des joueurs (la liste des partenaires n'est active que lors d'un jeu à 5 joueurs) et la liste des contrats avec les contrats possibles :

```
public PagePartie(Partie p)
{
     InitializeComponent();
     partie = p;
     var joueurs = p.Lister();
     preneur.ItemsSource = joueurs;
     if(joueurs.Length>4)
     {
       partenaire.ItemsSource = p.Lister();
       partenaire.IsEnabled = true;
     }
     contrats.ItemsSource = partie.Contrats;
}
```

Il faut encore rajouter cette opération qui va afficher le nombre choisi par le `Slider` :

```
private void NombreBouts(object sender, ValueChangedEventArgs
     e)
{
     nbBouts.Text = ((int)(e.NewValue)).ToString();
}
```

Il ne reste plus qu'à faire l'opération de validation qui va créer un tour de jeu. Il faut donc compléter la classe `Tour` pour qu'elle contienne les propriétés et opérations nécessaires, notamment le calcul du score. Différents tests doivent être effectués (des exceptions sont levées en cas de problème).

```
public class Tour
{
      private Contrat contrat;
      private Joueur preneur;
      private Joueur partenaire;
      private int nbJoueurs;
      private int nbOudlers=0;
      private int score=0;
      private bool petit=false;
      private bool poignee=false;

      public int NbOudlers { get => nbOudlers;
         set => nbOudlers = value; }
      public int Score { get => score; set => score = value; }
      public bool Petit { get => petit; set => petit = value;
      }
      public bool Poignee { get => poignee;
         set => poignee = value; }

      public Tour(int nb)
      {
         nbJoueurs = nb;
      }

      public void ChoixContrat(Contrat c)
      {
         contrat = c ?? throw new
      Exception(Textes.Strings.ContralNul);
      }

      public void ChoixPreneur(Joueur j)
      {
         preneur = j ?? throw new
      Exception(Textes.Strings.PreneurNul);
      }

      public void ChoixPartenaire(Joueur j)       {
         if (nbJoueurs < 5 && j != null)
            throw new Exception(Textes.Strings.Pas5);
         if (nbJoueurs == 5 && j == null)
            throw new Exception(Textes.Strings.PartenaireNul);
         partenaire = j;
      }

      public int CalculeScore()
      {
         if (preneur == null)
            throw new Exception(Textes.Strings.PreneurNul);
```

```
        if (nbJoueurs == 5 && partenaire == null)
          throw new Exception(Textes.Strings.PartenaireNul);
        if (contrat == null)
          throw new Exception(Textes.Strings.ContralNul);
        if (score == 0)
          throw new Exception(Textes.Strings.ScoreNul);
        int pointsAFaire=56;
        switch(nbOudlers)
        {
          case 1: pointsAFaire = 51;break;
          case 2:pointsAFaire = 41;break;
          case 3:pointsAFaire = 36;break;
        }
        int ecart = score - pointsAFaire;
        int total = (25 + Math.Abs(ecart) + (petit?10:0) ) *
    contrat.Coefficient + (poignee?10:0);
        if (ecart<0) total = -total;
          return total;
    }
}
```

Dans la classe `Partie`, il faut mettre en attribut une liste de tours :

```
private List<Tour> tours = new List<Tour>();
```

Il faut également proposer une opération pour ajouter un tour, et automatiquement sauvegarder la partie :

```
public void NouveauTour(Tour t)
{
    tours.Add(t);
    stockage.Sauver(this);
}
```

Il ne reste qu'à placer le code dans la fenêtre de partie pour valider un tour, ce qui récupère les données saisies, crée le tour et le joint à la partie, puis passe automatiquement au 2ème onglet. Le score est (pour l'instant) affiché à l'écran pour tester le calcul.

```
private void Valider(object sender, EventArgs e)
{
    try
    {
      Tour t = new Tour(partie.Lister().Length);
      t.ChoixContrat(contrats.SelectedItem as Contrat);
      t.ChoixPartenaire(partenaire.SelectedItem as Joueur);
      t.ChoixPreneur(preneur.SelectedItem as Joueur);
```

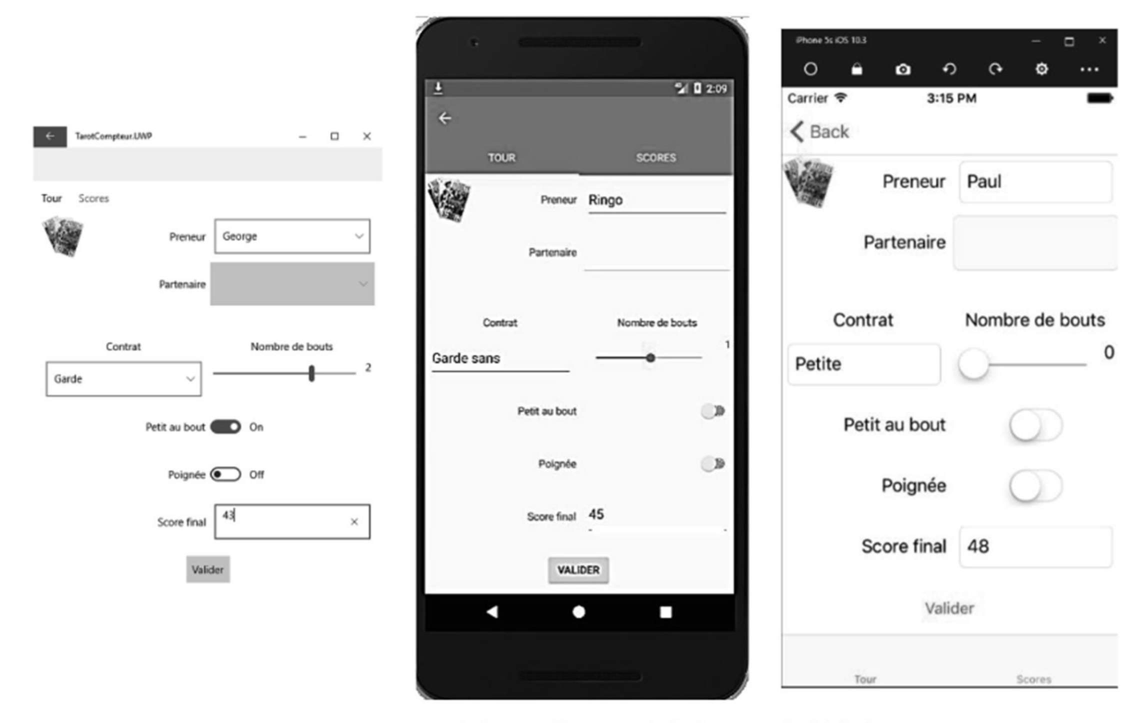
```
        t.NbOudlers = (int)bouts.Value;
        t.Petit = petit.IsToggled;
        t.Poignee = poignee.IsToggled;
        t.Score = Convert.ToInt32(score.Text);
        partie.NouveauTour(t);

        CurrentPage = Children[1];

        int calcul = t.CalculeScore();
        DisplayAlert("test", calcul.ToString(), "ok");

        // et on efface tout...
        contrats.SelectedItem = null;
        partenaire.SelectedItem = null;
        preneur.SelectedItem = null;
        petit.IsToggled = false;
        poignee.IsToggled = false;
        score.Text = "";
    }
    catch(Exception x)
    {
        DisplayAlert(Textes.Strings.Erreur, x.Message, "Ok");
    }
}
```

Le test du premier onglet donne les écrans suivants :



Capture 91 : tour de jeu, atelier tarot (windows, android, ios)

Pour le deuxième onglet, nous allons afficher les scores des différents joueurs, tour par tour. Il faut donc créer une liste (`ListView`) contenant différentes colonnes (une par joueur), le nom des joueurs en première ligne, une ligne par tour de jeu avec les scores de chaque joueur.

Il faut déjà ajouter dans la classe `Tour` une fonction qui calcule le score d'un joueur :

```
public int CalculeScore(Joueur j)
{
     int total = CalculeScore();
     int score = 0;
     if(!j.EstEgal(preneur) && !j.EstEgal(partenaire)) //
     defense
     {
       score = -total;
     }
     else if(j.EstEgal(partenaire) && nbJoueurs==5)
     {
       score = total;
     }
     else if(j.EstEgal(preneur))
     {
       int nbOpposants;
       if (nbJoueurs < 5 || preneur.EstEgal(partenaire))
         nbOpposants = nbJoueurs - 1;
       else
         nbOpposants = nbJoueurs - 2;
       score = total * nbOpposants;
     }
     return score;
}
```

Nous avons donc dans la partie métier une opération pour lister les joueurs (`Partie.Lister`), une opération pour calculer le score de chaque joueur (`Tour.CalculeScore`), auxquelles il faut ajouter dans la classe `Partie` une opération pour lister les tours de jeu :

```
public Tour[] Tours { get => tours.ToArray(); }
```

Il faut également placer dans la classe `Joueur` la gestion des scores, c'est-à-dire une propriété `Score` :

```
private int score=0;
public int Score {
```

```
        get => score;
        set => score = value;
}
```

Passons à la partie IHM…

Xamarin.Forms ne possède pas de composant pour dessiner facilement une table. Pour afficher les scores, nous allons donc ruser un peu…

Commençons par utiliser une disposition générale de type `StackLayout` :

```
<ContentPage Title="{x:Static resx:Strings.Scores}" >
<StackLayout Orientation="Vertical">
```

Pour les en-têtes, une grille en prévoyant la possibilité de cinq joueurs (donc 5 colonnes), en utilisant le *Data Binding*[1] pour se lier au modèle :

```
<Grid Margin="10">
<Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="*"/>
</Grid.ColumnDefinitions>
<Label Grid.Column="0" Text="{Binding Joueurs[0]}"
        HorizontalOptions="Center" FontAttributes="Bold" />
<Label Grid.Column="1" Text="{Binding Joueurs[1]}"
        HorizontalOptions="Center" FontAttributes="Bold" />
<Label Grid.Column="2" Text="{Binding Joueurs[2]}"
        HorizontalOptions="Center" FontAttributes="Bold" />
<Label Grid.Column="3" Text="{Binding Joueurs[3]}"
        HorizontalOptions="Center" FontAttributes="Bold" />
<Label Grid.Column="4" Text="{Binding Joueurs[4]}"
        HorizontalOptions="Center" FontAttributes="Bold" />
</Grid>
```

Pour les scores, nous allons utiliser la faculté des `ListView` de changer la forme de l’affichage. Notre modèle d’affichage va utiliser de nouveau une grille à cinq colonnes, avec des étiquettes liées au modèle :

```
<ListView x:Name="lesScores"
        HorizontalOptions="FillAndExpand" VerticalOptions="Fill"
```

[1] Liaison entre les données et l’IHM, utilisation du *pattern* MVVM avec Xamarin.Forms.

```
     Margin="10" BackgroundColor="Beige">
<ListView.ItemTemplate><DataTemplate><ViewCell >
<Grid> <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="*"/>
     </Grid.ColumnDefinitions>
     <Label Grid.Column="0" Text="{Binding Score[0]}"
     HorizontalOptions="Center" />
     <Label Grid.Column="1" Text="{Binding Score[1]}"
     HorizontalOptions="Center" />
     <Label Grid.Column="2" Text="{Binding Score[2]}"
     HorizontalOptions="Center" />
     <Label Grid.Column="3" Text="{Binding Score[3]}"
     HorizontalOptions="Center" />
     <Label Grid.Column="4" Text="{Binding Score[4]}"
     HorizontalOptions="Center" />
</Grid>
</ViewCell></DataTemplate></ListView.ItemTemplate>
</ListView>
```

Pour terminer, une nouvelle grille à cinq colonnes pour l'affichage des totaux :

```
        <Grid Margin="10" >
          <Grid.ColumnDefinitions>
          <ColumnDefinition Width="*"/>
          <ColumnDefinition Width="*"/>
          <ColumnDefinition Width="*"/>
          <ColumnDefinition Width="*"/>
          <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <Label Grid.Column="0" Text="{Binding Totaux[0]}"
     HorizontalOptions="Center" FontAttributes="Bold" />
        <Label Grid.Column="1" Text="{Binding Totaux[1]}"
     HorizontalOptions="Center" FontAttributes="Bold" />
        <Label Grid.Column="2" Text="{Binding Totaux[2]}"
     HorizontalOptions="Center" FontAttributes="Bold" />
        <Label Grid.Column="3" Text="{Binding Totaux[3]}"
     HorizontalOptions="Center" FontAttributes="Bold" />
        <Label Grid.Column="4" Text="{Binding Totaux[4]}"
     HorizontalOptions="Center" FontAttributes="Bold" />
     </Grid>
</StackLayout> </ContentPage>
```

Le composant `ListView` utilisant le modèle MVVM, il faut créer des classes de type « vue-modèle » pour lier la vue (le fichier XAML) avec le modèle (les données). Notre vue va devoir afficher trois listes :

- Les noms des joueurs (5 chaînes maximum)
- Les tours de jeu, chaque tour comprenant un score par joueur (5 entiers maximum)
- Les scores totaux (5 maximum)

Nous allons décomposer le « vue-modèle » en deux classes, une pour la partie score (un simple tableau de 5 entiers) :

```
public class ScoresViewModel
{
     private int?[] scores = new int?[5];
     public int?[] Score { get => scores; set => scores =
     value; }
}
```

Puis la classe « vue-modèle » pour l'ensemble de la vue. Cette classe doit implémenter `INotifyPropertyChanged` pour permettre d'indiquer à la vue qu'elle doit rafraîchir l'affichage.

```
public class PartieViewModel : INotifyPropertyChanged {
     public event PropertyChangedEventHandler
     PropertyChanged;

     private ObservableCollection<string> joueurs = new
     ObservableCollection<string>();
     public ObservableCollection<string> Joueurs
     { get => joueurs; }
     public void NomJoueur(int i, string nom)   {
       joueurs[i] = nom;
       PropertyChanged(this, new
     PropertyChangedEventArgs("Joueurs"));
     }
     private ObservableCollection<int?> totaux = new
     ObservableCollection<int?>();
     public ObservableCollection<int?> Totaux { get =>
     totaux; }
     public void TotalJoueur(int i, int value)  {
       totaux[i] = value;
       PropertyChanged(this, new
     PropertyChangedEventArgs("Totaux"));
     }
```

```
    private ObservableCollection<ScoresViewModel> txtScores
    = new ObservableCollection<ScoresViewModel>();
    public ObservableCollection<ScoresViewModel> TxtScores
    { get => txtScores; set => txtScores = value; }

    public PartieViewModel()       {
           for (int i = 0; i < 5; i++) {
               joueurs.Add(null);
               totaux.Add(null);
           }
    }
}
```

Dans le code de la classe de fenêtre partie, il faut ajouter un attribut représentant le « vue-modèle » :

```
private PartieViewModel viewModel = new PartieViewModel();
```

Au tout début du constructeur, lier le « vue-modèle » avec la vue :

```
BindingContext = viewModel;
```

Toujours dans le constructeur, initialiser les noms des joueurs et la liste des scores :

```
for (int i = 0; i < joueurs.Length; i++)
    viewModel.NomJoueur(i,joueurs[i].Nom);
lesScores.ItemsSource = viewModel.TxtScores;
```

Ajoutons une opération pour initialiser les tours de jeu dans le « vue-modèle » :

```
private void AjouteTour(Tour t, Joueur[] joueurs)
{
    ScoresViewModel scoresViewModel = new ScoresViewModel();
    for (int i = 0; i < joueurs.Length; i++)
    {
      int score = t.CalculeScore(joueurs[i]);
      scoresViewModel.Score[i] = score;
    }
    viewModel.TxtScores.Add(scoresViewModel);
}
```

Dans le constructeur cette opération sera appelée pour afficher les tours de jeu déjà existants (en cas de reprise de partie) :

```
foreach(Tour t in partie.Tours) {
     AjouteTour(t, joueurs);
}
```

Il faut encore une opération pour initialiser les totaux dans le vue-modèle, opération qui sera elle aussi appelée dans le constructeur :

```
private void AjouteTotaux(Joueur[] joueurs) {
     for (int i = 0; i < joueurs.Length; i++)
     {
       viewModel.TotalJoueur(i, joueurs[i].Score);
     }
}
```

Il ne reste plus qu'à modifier la fonction de validation, déjà commencée page 331, pour ajouter le code nécessaire au calcul des scores :

```
Joueur[] joueurs = partie.Lister();
ScoresViewModel scoresViewModel = new ScoresViewModel();
for(int i=0;i<joueurs.Length;i++)
{
     int score = t.CalculeScore(joueurs[i]);
     joueurs[i].Score += score;
}
AjouteTour(t,joueurs) ;
AjouteTotaux(joueurs) ;
CurrentPage = Children[1];
```

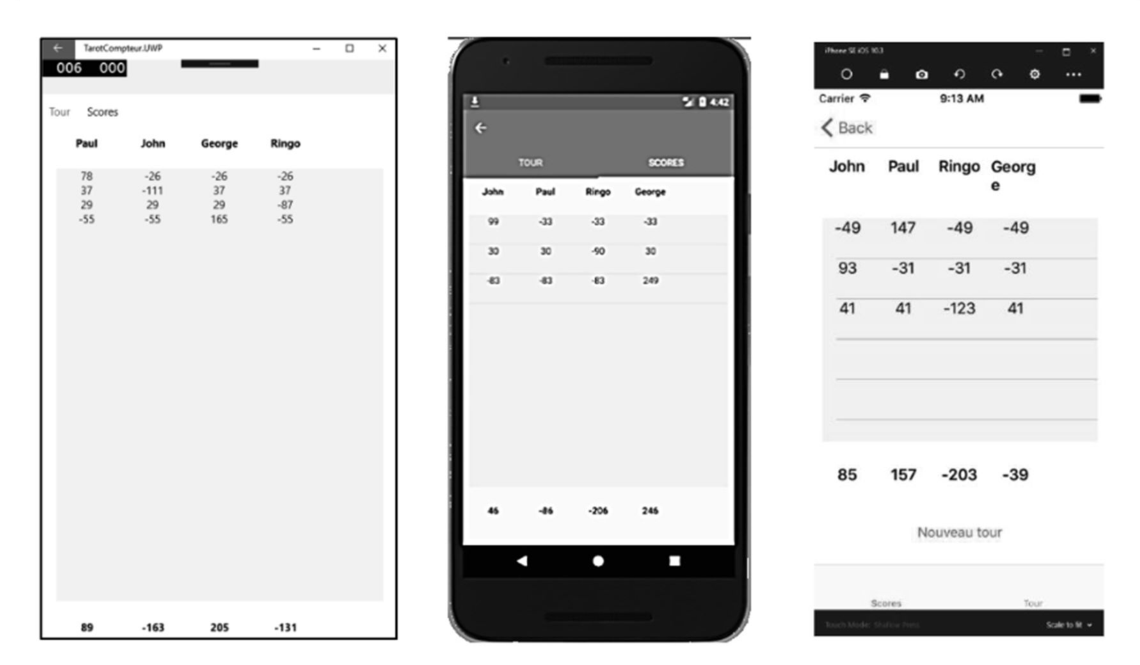
Cette partie est un peu complexe, notamment à cause du modèle MVVM utilisé par le composant `ListView` : ce modèle est très puissant mais un peu compliqué et très « verbeux »…

Il parait plus intéressant de commencer par l'onglet « scores » : il suffit alors d'échanger les deux onglets dans le XAML, ou alors de modifier le constructeur pour que l'objet de départ soit le deuxième :

```
CurrentTab = Children[1] ;
```

Notre application est presque terminée… il ne reste plus qu'à gérer la sauvegarde des données !

Le test donne les images suivantes :



Capture 92 : résumé d'une partie de tarot (windows, android, ios)

j) Persistance des données

Nous avons vu au chapitre VII comment les données pouvaient être sauvegardées sur un téléphone. L'utilisation d'une base de données locale semble ici superflue (il y a peu de données à sauvegarder). Nous avons le choix entre la gestion des paramètres (voir VII.1.d) ou la sauvegarde d'un fichier (VII.2). Dans tous les cas, il faut que nos classes modèle (celles qui stockent les données) soient sérialisables.

Commençons par modifier la classe `Partie`. Celle-ci doit être marquée comme étant un `DataContract` ce qui la rend sérialisable.

```
[DataContract]
public class Partie
```

Parmi ses attributs, les données doivent être marquées comme `DataMember` :

```
[DataMember]
private List<Joueur> joueurs = new List<Joueur>();
[DataMember]
private Contrat[] contrats = new Contrat[4];
[DataMember]
private List<Tour> tours = new List<Tour>();
```

Il faut donc que la classe Joueur soit sérialisable, de même que les classes Tour et Contrat.

Passons à la classe Joueur : elle doit aussi être un DataContract :

```
[DataContract]
public class Joueur
```

Le nom et le score du joueur doivent tous deux être des DataMember :

```
[DataMember]
private string nom;
[DataMember]
private int score=0;
```

La classe Contrat doit aussi être sérialisable :

```
 [DataContract]
public class Contrat
```

Ses deux attributs sont tous deux sauvegardables :

```
 [DataMember]
private int coefficient;
 [DataMember]
private string type;
```

Il ne reste plus que la classe Tour : elle aussi est un DataContract :

```
 [DataContract]
public class Tour
```

Tous ses attributs sont à sauvegarder, donc doivent être marqués DataMember :

```
        [DataMember]
        private Contrat contrat;
        [DataMember]
        private Joueur preneur;
        [DataMember]
        private Joueur partenaire;
        [DataMember]
        private int nbJoueurs;
        [DataMember]
        private int nbOudlers=0;
        [DataMember]
        private int score=0;
```

```
        [DataMember]
        private bool petit=false;
        [DataMember]
        private bool poignee=false;
```

Il faut choisir un mode de sauvegarde… Prenons les paramètres, ce qui est le plus simple, et utilisons la sauvegarde sous forme d'une chaîne de caractères JSON (nous pourrions utiliser XML, mais JSON est plus compact donc occupe moins d'espace).

Nous allons implémenter la sauvegarde de la partie dans une classe. Cette classe devra implémenter l'interface de sauvegarde définie dans la couche métier. Elle va utiliser un `DataContractJsonSerializer` pour sérialiser une `Partie` et les paramètres du système pour le stockage. Comme vu page 149, il faut installer un *plugin* pour pouvoir utiliser les paramètres de manière multiplateforme. Une fois ce *plugin* installé (dans tous les projets de la solution) la classe peut être saisie :

```
class SauveParams : ISauvegarde
{
    public Partie Charger()
    {
      Partie p;
      string json =
    CrossSettings.Current.GetValueOrDefault("Partie", "");
      if(json=="")
       p = new Partie(this);
      else
      {
        DataContractJsonSerializer ser = new
    DataContractJsonSerializer(typeof(Partie));
        using (MemoryStream flux = new
    MemoryStream(Encoding.UTF8.GetBytes(json)))
        {
          p = ser.ReadObject(flux) as Partie;
          p.Stockage = this;
        }
      }
      return p;
}

public void Sauver(Partie p)
    {
      DataContractJsonSerializer ser = new
    DataContractJsonSerializer(typeof(Partie));
```

```
        using (MemoryStream flux = new MemoryStream())
        {
            ser.WriteObject(flux,p);
          string result =
      Encoding.UTF8.GetString(flux.ToArray(),
      0,(int)flux.Length);
          CrossSettings.Current.AddOrUpdateValue("Partie",
      result);
        }
      }
}
```

Il faut à présenter remplacer la classe factice de persistance par une instance de celle-ci (constructeur de la page d'accueil) :

```
public MainPage() {
     InitializeComponent();
     var assembly = typeof(MainPage).GetTypeInfo().Assemby ;
     image.Source =
     ImageSource.FromResource("TarotCompteur.Images.tarot.png
     ");
     stockage = new SauveParams();
     jeu = new Tarot(stockage);
}
```

Il faut encore rajouter dans la classe `Partie` une opération pour la sauvegarder :

```
        public void Sauver()
        {
            stockage.Sauver(this);
        }
```

Enfin, dans la fenêtre où l'on valide un tour, ne pas oublier de sauvegarder la partie (fonction de validation du tour) :

```
partie.Sauver();
CurrentPage = Children[0];
```

Testez : tout doit fonctionner !

k) Améliorations possibles

Cette application est volontairement limitée pour pouvoir être décrite totalement en peu de pages… La première version que j'ai déposée sur les différents magasins d'applications est celle décrite dans ces pages. Il est

possible que je réalise des mises à jour de l'application, vous pouvez quant à vous essayer de l'améliorer en ajoutant des fonctionnalités, en modifiant l'IHM, etc…

Voici quelques idées de modifications intéressantes :

- Gérer le chelem
- Inclure la possibilité de jouer une misère (personne ne prend → points inversés)
- Permettre d'avoir plusieurs parties en mémoire, de sauvegarder le contenu d'une partie…

7. Le pendu

Difficulté : moyen

Plateformes ciblées : Windows 10, Android, iOS

Points techniques abordés : fichiers ressources (textes et images), IHM simple

Langages utilisés : C++

Outils utilisés : Qt Creator

Téléchargement de l'application :

- Pour Android : https://frama.link/Pn0VwT7n
- Pour Windows 10 : https://frama.link/XxpQv-Eg
- Pour iOS : https://frama.link/nS2NgRym

Téléchargement des codes sources : https://frama.link/sw915Qav

a) Description de l'application souhaitée

Le jeu du « pendu » est un classique des jeux de lettres : un mot quelconque est tiré au hasard mais le joueur ne voit que le nombre de lettres du mot. Il doit donc proposer des lettres : si celles-ci font partie du mot, elles apparaissent et permettent de construire petit à petit celui-ci, sinon un personnage de « pendu » est dessiné petit à petit (10 étapes) : 10 erreurs et le joueur a perdu (il est pendu).

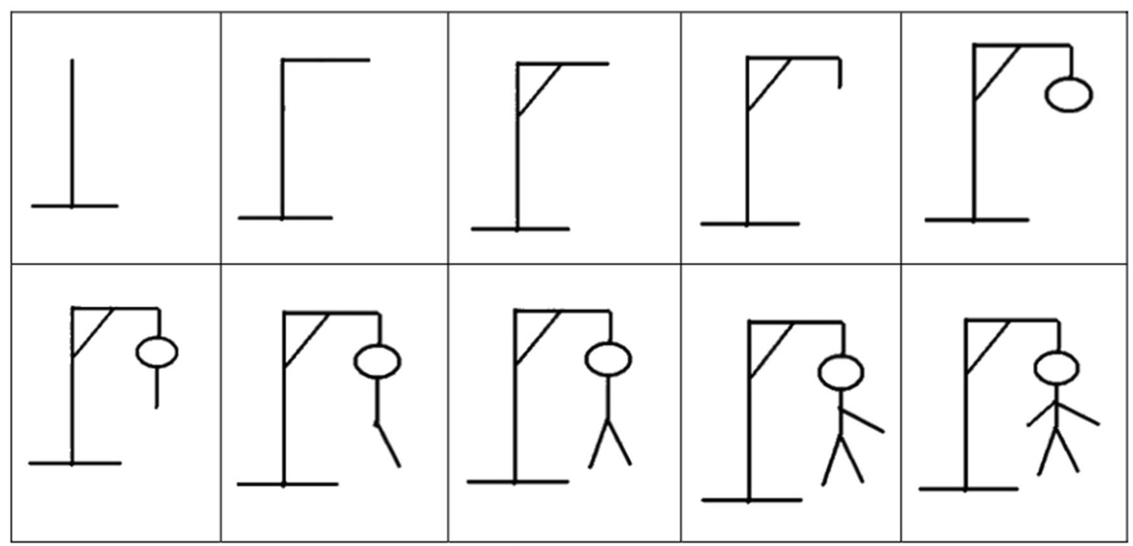


Tableau 13 : les 10 étapes du "pendu"

b) **Conception de l'application : *design***

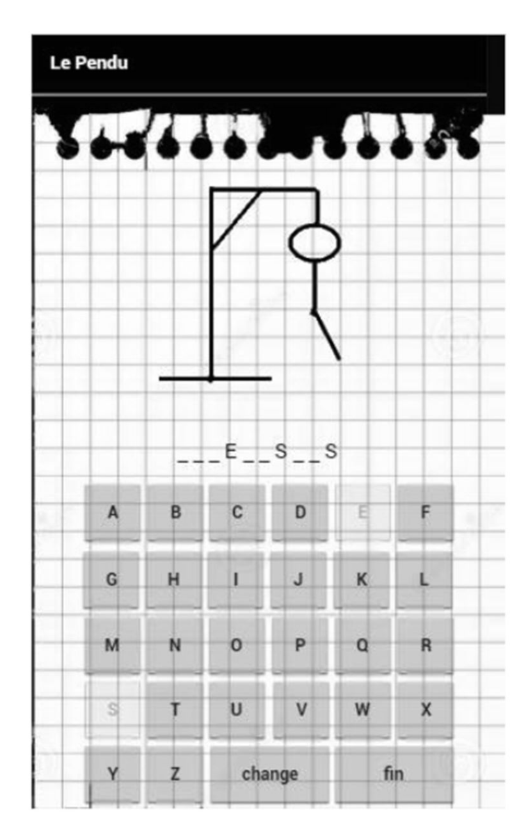
Cette application n'a besoin que d'un seul écran qui va comporter le mot à trouver, l'image du « pendu » ainsi que les lettres à choisir. La seule interaction de l'utilisateur étant le choix d'une lettre, le plus simple est de définir 26 boutons (un par lettre) disposés en grille. L'orientation sera « portrait » (plus simple avec un téléphone). Pour donner un petit aspect croquis, l'image de fond peut être celle d'un bloc-notes, le côté « papier » donnant un aspect sympathique à l'application.

Les 26 boutons représentant les lettres sont ici disposés en grille ; à chaque fois qu'un bouton est pressé, il sera désactivé puisque la lettre a déjà été jouée. Le dessin sera mis à jour en cas d'erreur.

Le bouton « fin » peut servir à abandonner la partie.

Le bouton « change » peut servir à changer de mot.

Une fois une partie finie (le mot est trouvé ou le joueur est pendu) il sera proposé de rejouer.



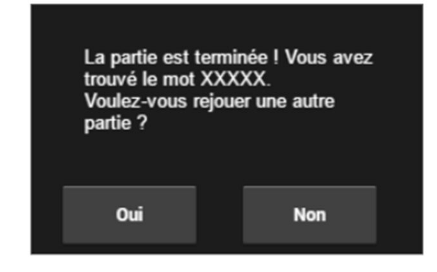


Diagramme 20 : maquette de l'application "pendu"

c) Conception : architecture

L'architecture de cette application sera classique :

- Une couche s'occupera de la partie « IHM » : l'affichage du dessin du pendu, le placement des boutons, l'affichage du mot à trouver, l'interaction avec le joueur
- Une couche s'occupera de la partie « métier » : la règle du jeu, le tirage d'un mot au hasard
- Une couche s'occupera de la partie « ressources » : le dictionnaire (les mots possibles) sera dans un fichier texte lié à l'application

Cette architecture peut se représenter par le diagramme de paquetages suivant :

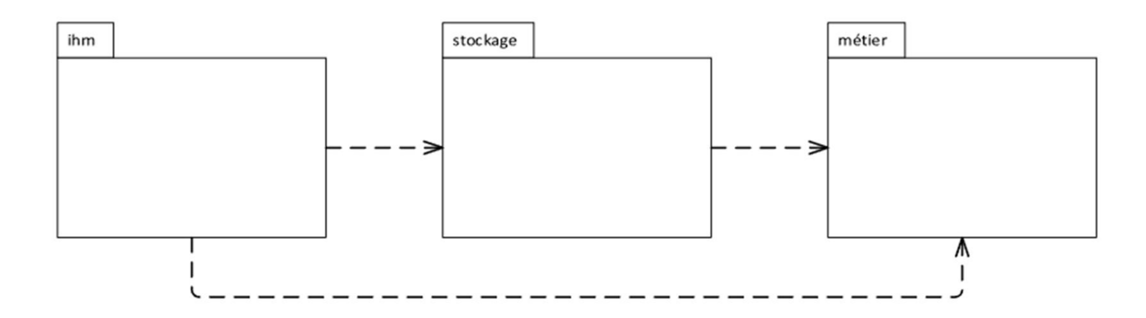


Diagramme 21 : diagramme de packages de l'application pendu

d) Conception : paquet « IHM »

Ce paquet comprendra une classe pour l'écran principal ; il sera lié à la couche de stockage et à la couche métier. Cette classe contiendra les évènements liés à l'interaction avec l'utilisateur : le clic sur un bouton « lettre », le clic sur le bouton « fin » et le clic sur le bouton « change ».

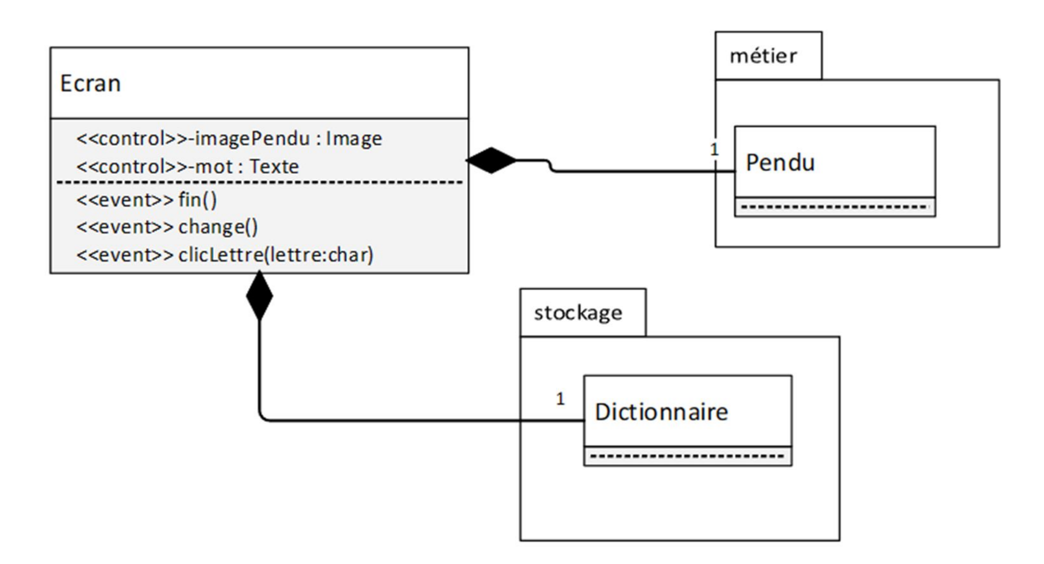


Diagramme 22 : ihm de l'application pendu

e) Conception : paquet « métier »

Ce paquet comprend une classe qui permet de gérer le jeu ainsi qu'une interface pour le lien avec la couche de stockage (le fournisseur de mots).

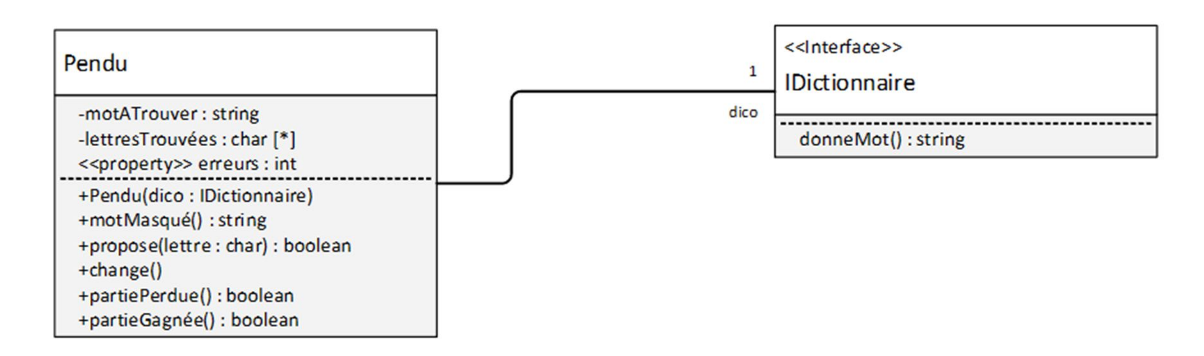


Diagramme 23 : diagramme de classes de la partie métier du pendu

L'interface `IDictionnaire` servant de couche d'abstraction à la partie « stockage », elle ne comporte qu'une seule opération destinée à fournir un mot au hasard. La classe `Pendu` doit stocker le mot à trouver, les lettres déjà trouvées par le joueur, le nombre d'erreurs effectuées. Elle doit fournir le mot sous une forme masquée (les lettres non trouvées remplacées par des _), indiquer la lettre proposée par l'utilisateur, et permettre le changement de mot (et donc la réinitialisation des informations). Il faut aussi indiquer si la partie est gagnée ou perdue.

f) Développement de l'application : métier

Commençons par créer une application « Qt Widgets » avec Qt Creator.

Nous allons débuter le développement par la couche métier. L'interface du dictionnaire sera déclarée sous forme d'une classe ne comportant que des méthodes virtuelles pures[1] publiques:

```cpp
#include <string>
class IDictionnaire {
public:
    virtual ~IDictionnaire() {}
    virtual std::string donneMot() const = 0;
}
```

Cette interface peut être implémentée dans l'objet factice suivant, ce qui permet de tester la couche métier indépendamment de la couche de stockage :

[1] La notion d'interface n'existe pas en C++ mais une classe peut être utilisée.

```
#include "idictionnaire.h"

class DicoFactice : public IDictionnaire
{
public:
    std::string donneMot() const {return "PENDU";}
};
```

Il reste à créer la classe `Pendu` ; commençons par définir ses attributs : une chaîne pour le mot à trouver, un ensemble de caractères pour les lettres déjà trouvées, un simple entier pour le nombre d’erreurs, un pointeur sur l’interface de dictionnaire pour l’association. Un accesseur sur le nombre d’erreurs sera en outre utile :

```
#include "idictionnaire.h"
#include <string>
#include <set>
class Pendu
{
private:
    std::string motATrouver;
    std::set<char> lettresTrouvees;
    int erreurs;
    IDictionnaire* dico;
public:
    Pendu(IDictionnaire* dico_);
    int Erreurs() const {return erreurs;}
    std::string motMasque() const;
    bool propose(char lettre);
    void change();
    bool partiePerdue() const ;
    bool partieGagnee() const ;
     std::string mot() const {return motATrouver;}
};
```

Le constructeur de cette classe initialise les attributs, notamment l’association :

```
Pendu::Pendu(IDictionnaire *dico_) : dico(dico_), erreurs(0)
{
     motATrouver = dico->donneMot() ;
}
```

Pour construire le mot masqué, il suffit de parcourir le mot à trouver et d’utiliser l’ensemble des lettres trouvées :

```
std::string Pendu::motMasque() const
{
    std::string mot="";
    for(char lettre : motATrouver)
    {
        if(lettresTrouvees.count(lettre))
            mot+=lettre;
        else
            mot+='_';
       mot += ' ' ;
    }
    return mot;
}
```

La proposition d'une lettre va éventuellement augmenter le nombre d'erreurs, et modifier l'ensemble des lettres trouvées…

```
bool Pendu::propose(char lettre)
{
    bool ok=false;
    if(motATrouver.find(lettre)!=std::string::npos)
    {
        lettresTrouvees.insert(lettre);
        ok=true;
    }
    else
        ++erreurs;
    return ok;
}
```

La fonction `change` se contente de réinitialiser les attributs et de redemander un mot au dictionnaire lié :

```
void Pendu::change()
{
    erreurs = 0;
    lettresTrouvees.clear();
    motATrouver = dico->donneMot();
}
```

Pour savoir si une partie est perdue, il suffit de compter le nombre d'erreurs :

```
bool Pendu::partiePerdue() const
{
    return erreurs>=10;
}
```

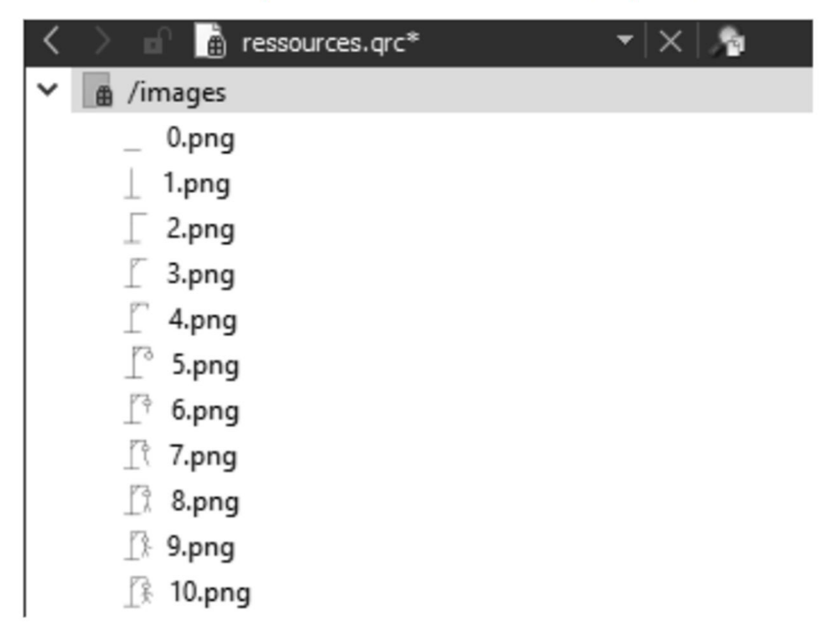
La partie est gagnée si toutes les lettres du mot sont trouvées et que le mot masqué ne contient plus de '_' :

```
bool Pendu::partieGagnee() const
{
    return motMasque().find('_')==std::string::npos;
}
```

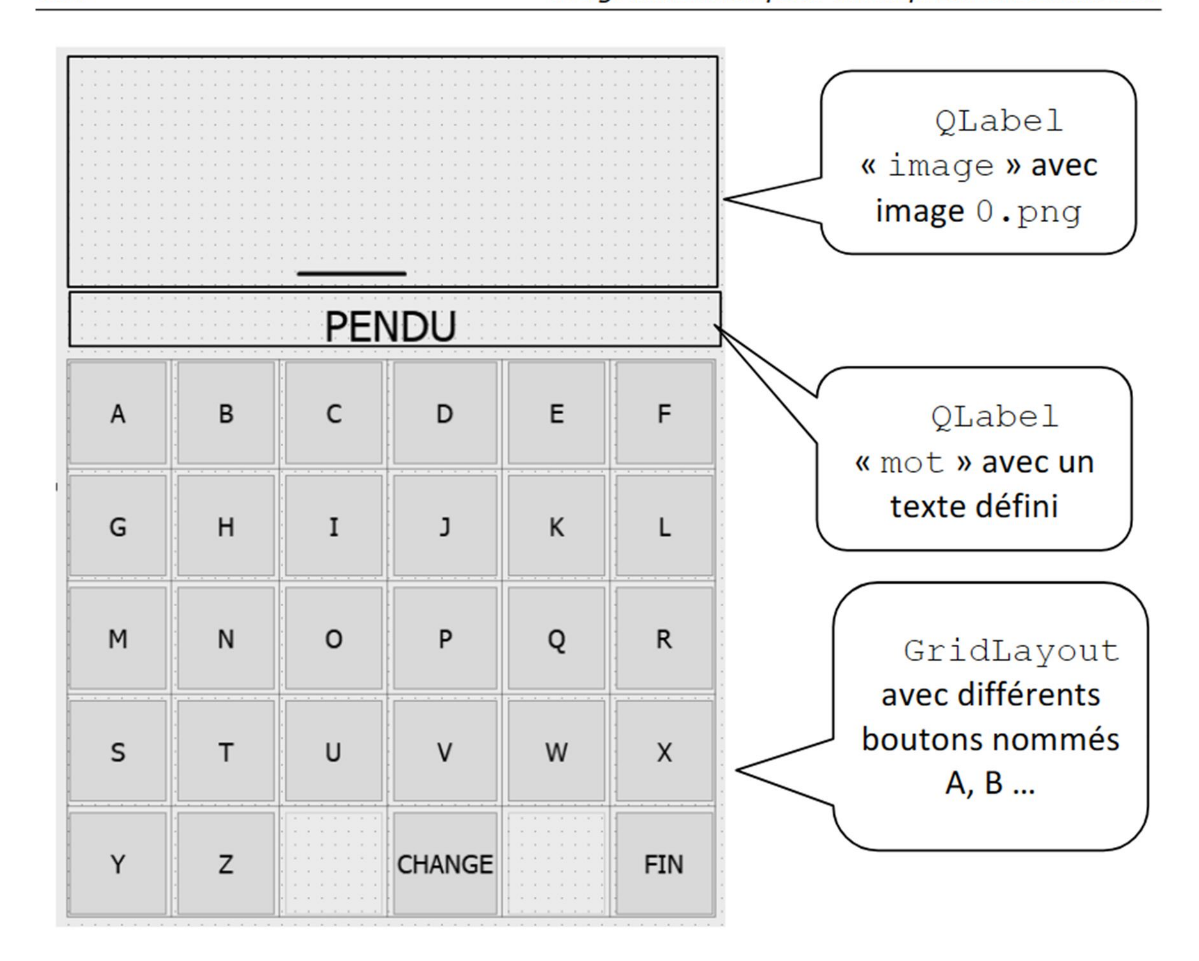
g) Développement : couche IHM

La couche IHM comprend une fenêtre de type `MainWindow`. Sa disposition générale est un *Vertical Layout*. Il comprend un `QLabel` pour afficher l'image (les 10 images seront prises dans les ressources comme vu page 141), un autre `QLabel` pour afficher le mot masqué, un `GridLayout` pour les 28 boutons.

Pour placer les 10 images dans les ressources, il faut ajouter au projet un fichier de type « ressources Qt ». Sous le préfixe « `images` », placer les 10 fichiers PNG correspondant aux 10 états du pendu :



Au départ, le `QLabel` utilisé pour l'image est initialisé avec l'image 0 (aucune erreur).



Pour l'image de fond, il faut utiliser du code dans le constructeur (l'image est elle aussi présente dans les ressources) dans un format rappelant les feuilles de styles CSS :

```
setStyleSheet("#centralWidget {background-image :
    url(\":/images/feuille.png\") 0 0 0 0 strech
    strech} ;");
```

Dans la classe de fenêtre, on ajoute les attributs nécessaires pour le lien avec les couches métier et stockage :

```
private:
    IDictionnaire* dico;
    Pendu* pendu;
```

Ces deux attributs doivent être initialisés dans le constructeur (pour l'instant, nous initialiserons le dictionnaire avec la version de test) ainsi que le mot masqué :

```
dico = new DicoFactice();
pendu = new Pendu(dico);
ui->mot->setText(QString::fromStdString(pendu->motMasque()));
```

Ne pas oublier de libérer[1] ces deux pointeurs dans le destructeur de la fenêtre :

```
delete pendu;
delete dico;
```

Il faut ensuite créer pour chaque bouton un signal relié au *slot* `Clicked()`. Cette partie est un peu fastidieuse mais la seule alternative serait de ne pas créer les boutons dans le *designer* mais par code, dans une boucle. Restons simple, il ne s'agit que de 26 fonctions après tout…

Pour chaque signal créé, l'opération devra proposer à la couche métier la lettre, mettre à jour le mot masqué et l'image, tester la fin de la partie… il est donc intéressant de factoriser le tout dans une opération privée de la classe :

```
void MainWindow::nouvelleLettre(char c)
{
    bool ok = pendu->propose(c);
    if(!ok) // une erreur de plus
    {
        changeImage(pendu->Erreurs());
        if(pendu->partiePerdue())
            perdu();
    }
    else // une lettre de plus
    {
        ui->mot->setText(QString::fromStdString(
     pendu->motMasque()));
        if(pendu->partieGagnee())
            gagne();
    }
}

void MainWindow::changeImage(int erreurs)
{
    QString file =
     ":/images/"+QString::number(erreurs)+".png";
```

[1] En effet C++ ne libère pas automatiquement la mémoire allouée par `new`, il faut donc le faire dans le destructeur, appelé automatiquement à la fin du programme.

```
    QPixmap img(file);
    ui->image->setPixmap(img);
}
```

Chaque signal relié à un bouton est donc similaire au signal suivant (le bouton est désactivé et on indique la proposition d'une nouvelle lettre) :

```cpp
void MainWindow::on_A_clicked()
{
    ui->A->setEnabled(false);
    nouvelleLettre('A');
}
```

Quand l'utilisateur a perdu, il faut lui demander s'il veut recommencer ou quitter le jeu :

```cpp
void MainWindow::perdu()
{
    QString msg = "Vous n'avez pas trouvé le mot " +
     QString::fromStdString(pendu->mot())+
     ". Voulez-vous rejouer ?";
    int rep = QMessageBox::question(this,"Partie perdue"
     ,msg,QMessageBox::Yes, QMessageBox::No);
    if(rep==QMessageBox::Yes)
    {
        change();
    }
    else if(rep==QMessageBox::No)
    {
        this->close();
    }
}
```

De même quand il a gagné :

```cpp
void MainWindow::gagne() {
    QString msg = "Bravo ! Vous avez trouvé le mot "
     +QString::fromStdString(pendu->mot())+
     ". Voulez-vous rejouer ?";
    int rep = QMessageBox::question(this,"Partie gagnée"
     ,msg,QMessageBox::Yes, QMessageBox::No);
    if(rep==QMessageBox::Yes)
    {
        change();
    }
    else if(rep==QMessageBox::No)
    {
```

```
        this->close();
    }
}
```

La fonction de réinitialisation de la partie doit également remettre les boutons actifs, ce qui peut se faire dans une boucle (pour éviter les 26 lignes d'activation bouton par bouton) :

```
void MainWindow::change()
{
    pendu->change();
    changeImage(0);
    ui->mot->setText(QString::fromStdString(pendu-
     >motMasque()));
    for(int i=0;i<ui->centralWidget->children().count();i++)
    {
        QPushButton* button = dynamic_cast<QPushButton*>
     (ui->centralWidget->children().at(i));
        if(button)
            button->setEnabled(true);
    }
}
```

Lors du clic sur le bouton pour changer le mot, une confirmation est demandée à l'utilisateur (en cas de clic involontaire) :

```
void MainWindow::on_change_clicked()
{
    int rep = QMessageBox::question(this,"Recommencer",
     "Etes vous sûr de vouloir recommencer ?"
     ,QMessageBox::Yes, QMessageBox::No);
    if(rep==QMessageBox::Yes)
        change();
}
```

Et enfin, lors du clic sur le bouton de fin, une confirmation est également demandée :

```
void MainWindow::on_fin_clicked()
{
    int rep = QMessageBox::question(this,"Terminer",
     "Etes vous sûr de vouloir quitter le
     jeu?",QMessageBox::Yes, QMessageBox::No);
    if(rep==QMessageBox::Yes)
        close();
}
```

Il est à présent possible de tester l'application avec le dictionnaire factice.

h) Développement : couche de stockage

Dans cette partie nous allons nous intéresser à la gestion du dictionnaire. Nous allons donc utiliser un fichier texte contenant des mots : ce fichier servira de dictionnaire. Vous pouvez le créer vous-même (mais si vous voulez beaucoup de mots, ça sera long…) ou utiliser un fichier texte existant. J'utilise pour ma part un fichier assez complet (336 000 mots) que vous pouvez télécharger à l'URL suivante : https://frama.link/bnFn9zj8. Il faut commencer par placer ce fichier dans les ressources, par exemple sous le préfixe `/textes`. Il faut ensuite créer une classe qui implémente l'interface `IDictionnaire`.

Cette classe va contenir un tableau de mots, initialisé à partir des ressources. Il suffira de prendre une valeur au hasard dans ce tableau pour fournir un mot.

```cpp
class Dictionnaire : public IDictionnaire {
    std::vector<std::string> mots;
public:
    Dictionnaire();
    std::string donneMot() const;
};
```

```cpp
std::string Dictionnaire::donneMot() const {
    int i = std::rand()%mots.size();
    return mots[i];
}
```

Dans le constructeur, nous allons utiliser les fonctions d'entrée/sortie de Qt pour lire le fichier placé dans les ressources. En ne conservant que les mots entre 4 et 16 lettres (ces chiffres sont réglables), nous initialisons notre dictionnaire :

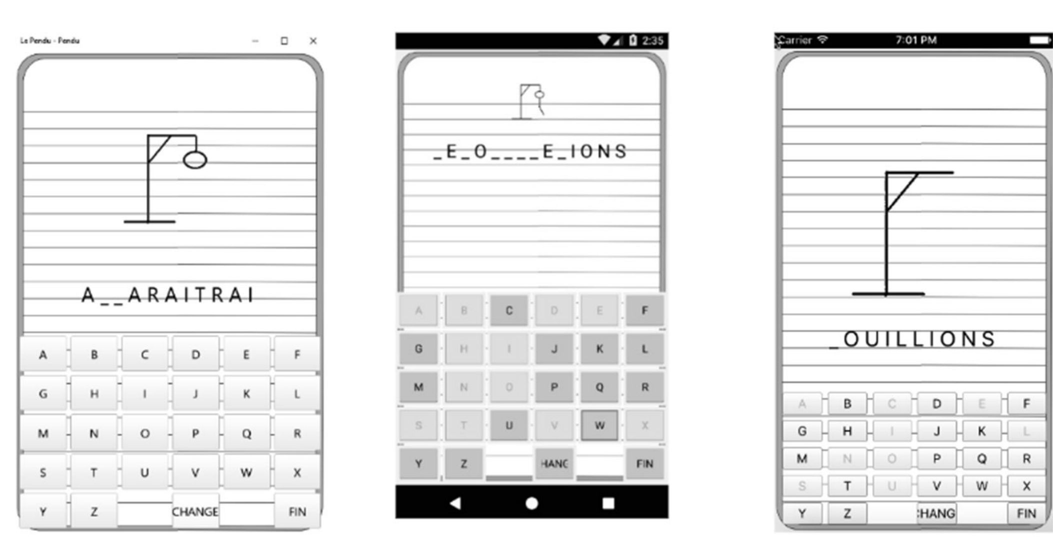
```cpp
Dictionnaire::Dictionnaire() {
    std::srand(std::time(nullptr));
    QFile file(":/textes/dico.txt");
    if(file.open(QIODevice::ReadOnly|QIODevice::Text)) {
        QTextStream flux(&file);
        while(!flux.atEnd()){
            QString mot = flux.readLine();
            if(mot.length()>=4 && mot.length()<=16)
```

```
                mots.push_back(mot.toStdString());
            }
            file.close();
        }
}
```

Il ne reste plus qu'à remplacer, dans la fenêtre principale, la création du dictionnaire factice par notre dictionnaire réel :

```
    dico = new Dictionnaire();
```

L'application est terminée ! Ne reste plus qu'à tester !



Capture 93 : application pendu sous Windows, Android, iOS

i) Améliorations possibles

Cette application est volontairement très simple et limitée pour pouvoir être décrite totalement en peu de pages… La première version que j'ai déposée sur les différents magasins d'applications est celle décrite dans ces pages. Il est possible que je réalise des mises à jour de l'application, vous pouvez quant à vous essayer de l'améliorer en ajoutant des fonctionnalités, en modifiant l'IHM, etc…

Voici quelques idées de modifications intéressantes :

- Utiliser un dictionnaire avec les définitions de manière à proposer la définition du mot à la fin (en cas de mot inconnu)
- Permettre l'utilisation de plusieurs dictionnaires avec choix de l'utilisateur

- Faire des niveaux de difficultés (mots plus ou moins complexes…)
- Utiliser des dictionnaires thématiques avec choix du thème
- Faire une IHM plus agréable

8. Convertisseur décimal-hexa-binaire

Difficulté : moyen

Plateformes ciblées : Android, iOS, Windows

Points techniques abordés : IHM dynamique, évènements

Langages utilisés : Javascript, HTML, CSS

Outils utilisés : Visual Studio, Cordova

Téléchargement de l’application :

- Pour Android : https://frama.link/_VUhB84u
- Pour Windows : https://frama.link/s0Ggk3fy
- Pour iOS : https://frama.link/ZAKg0fVd

Téléchargement des codes sources : https://frama.link/tWfK59Yq

a) Description de l’application souhaitée

Les programmeurs ont souvent besoin de réaliser des conversions de nombres entiers entre plusieurs bases, notamment la base 10 (la base naturelle pour nous), la base 2 (le binaire, base naturelle pour un ordinateur) et la base 16 (hexadécimal, proche du binaire et du décimal). Une petite application peut facilement effectuer ces calculs.

b) Conception de l'application



Figure 8 : maquette application convertisseur

L'IHM doit être très simple et ne comporter que peu de contrôles. Il faut des zones pour saisir les nombres et des zones pour afficher les conversions. Pour limiter les contrôles, il est pertinent de définir trois zones (une par base) servant à la fois à l'affichage et à la saisie : celle qui a le focus est en saisie, les autres en affichage. La saisie n'utilisera pas le clavier « flottant » du téléphone et des boutons seront déposés à cet effet. Il sera possible de choisir via des boutons radio le format (nombre de bits).

c) Développement : page HTML

L'application en Cordova comprend une page HTML par écran. Dans notre cas, il n'y a qu'un seul écran, donc toute l'interface sera codée dans le fichier `index.html`.

Tel que décrit au XII.8.b), cet écran contient 6 zones, disposées verticalement :

- Le titre
- La zone « base 10 » comprenant un texte et une zone de saisie
- La zone « base 16 », identique
- La zone « base 2 », identique
- La zone « choix du format » comprenant 2 boutons radio
- La zone « clavier » comprenant les touches à appuyer pour saisir les chiffres.

Chacune de ces zones sera donc, en HTML, une `div`.

La première est simple et ne contient que le titre :

```
<div><h1>Calculette pour programmeur</h1></div>
```



Les 3 zones suivantes étant similaires, il sera intéressant de leur donner la même classe. Dans chaque zone, nous aurons du simple texte (indiquant la base, une image peut aussi faire l'affaire), et une zone de saisie (balise `input`). Afin de permettre leur personnalisation par CSS et leur manipulation par *Javascript*, nous affecterons un identifiant (`id`) à chaque contrôle interactif et nous utiliserons les classes CSS pour agir sur l'affichage.

Les contrôles de saisie seront indiqués `readonly` car nous n'utiliserons pas le clavier du système mais notre propre clavier.

```
<div class="base">
    <label class="nombre focus">10</label>
    <input type="number" id="nbr10" placeholder="nombre
    décimal" class="saisie focus" readonly />
</div>
```

Dans le code précédent, la classe `base` est utilisée pour définir le style de chaque base (ensemble de contrôles liés à une base), la classe `nombre` sert à définir l'affichage des nombres de base (10, 16, 2), la classe `focus` à définir la modification d'affichage due à la prise du « focus », et enfin la classe `saisie` pour l'affichage des zones de saisie de nombre.

Les deux autres `div` sont similaires.

Pour le choix du format (octet, mot…) nous regrouperons un contrôle de type bouton radio avec un texte dans une balise `span` (de classe `format`), un identifiant étant donné à chaque bouton radio :

```
<div>
<span class="format">
      <input type="radio" name="format" value="byte"
    id="format8" checked>
      <label for="format8">Octet (8 bits)</label>
    </span>
    <span class="format">
      <input type="radio" name="format" id="format16"
    value="word" />
      <label for="format16">Mot (16 bits)</label>
    </span>
</div>
```

Enfin, pour le clavier de 16 touches, nous créerons par code celui-ci et par conséquent le fichier HTML ne contient que la `div`, identifiée :

```
<div id="clavier"></div>
```

d) Développement : styles CSS

Afin de personnaliser l'affichage, il faut modifier la feuille de style CSS (`index.css`) créée automatiquement.

La classe `base` (utilisée pour les contrôles relatifs à une base) doit être en conséquence modifiée, comme suit, pour avoir un affichage centré dans la page, un espacement minimum avec les autres sections, par exemple :

```css
.base {
    display: block;
    width: 90%;
    margin-left: auto;
    margin-right: auto;
    margin-top: 10px;
    margin-bottom: 20px;
    padding: 4px;
}
```

L'affichage d'un nombre de base (10, 16, 2) utilisera 10% de la largeur qui sera donc fixe :

```css
.nombre{
    display:inline-block;
    width:10%;
}
```

Pour la zone de saisie, occupant 80% de la largeur, nous allons l'entourer d'une bordure à coins arrondis (plus jolis) et utiliser une police un peu plus grande que celle par défaut :

```css
.saisie {
    display: inline-block;
    width: 80%;
    border: 1px solid black;
    border-radius: 3px;
    height:auto;
    font-size:18px;
}
```

Les cases à cocher et les boutons radio sont un peu petits par défaut, nous pouvons les augmenter facilement :

```css
input[type=checkbox]{
    transform:scale(2);
}
input[type=radio]{
    transform:scale(2);
}
```

Pour la zone qui obtient le focus, nous allons passer le texte en gras et l'entourer d'une bordure en pointillés légers :

```
.focus{
    font-weight:bold;
    border : 1px dashed gray;
}
```

e) Développement : gestion du focus

Les zones de saisie étant marquées en `readonly`, le système ne gère pas la prise de focus. Nous préférons ne pas utiliser le système du clavier pour permettre à l'utilisateur de ne saisir que les chiffres possibles suivant la base :

- De 0 à 9 en décimal
- De 0 à 9, A, B, C, D, E et F en hexadécimal
- Uniquement 0 et 1 en binaire

Il faut donc détecter l'évènement « `click` » sur chaque zone de saisie et attribuer le focus à sa ligne (son parent).

Pour simplifier l'écriture du code *Javascript*, la bibliothèque *JQuery*[1] sera utilisée.

La fonction suivante (placée dans `index.js`) sert à retirer le focus de toutes les sections puis à l'affecter à l'élément parent de l'élément passé en paramètre :

```
function putFocus(elt) {
     $(".base").removeClass("focus");
     $(elt.parentElement).addClass("focus");
}
```

Dans la fonction `onDeviceReady` (appelée quand l'application est chargée) il ne reste qu'à affecter à l'évènement `click` des 3 contrôles de saisie la fonction pour attribuer le focus :

```
$(".saisie").click(function (event) {
     putFocus(event.currentTarget); });
```

[1] *JQuery* est une bibliothèque *Javascript* gratuite permettant de faciliter l'accès aux éléments du document HTML depuis le code *Javascript*.

Un test rapide montre bien que le clic sur une zone de saisie lui transmet le focus :

f) Développement : création du clavier

Le clavier se compose de 16 boutons, chacun représentant un chiffre. Certains étant accessibles en binaire (0 et 1), d'autres en décimal (de 0 à 9) et les autres en hexa, il est intéressant de leur donner une classe afin de pouvoir les manipuler par lot :

- Classe « `base10` » pour les boutons actifs en décimal
- Classe « `base2` » pour les boutons actifs en binaire
- Classe « `base16` » pour les boutons actifs en hexadécimal

Les boutons seront placés dans une table HTML (4 lignes / 4 colonnes), à l'intérieur de la `div` « `clavier` ». Le clic sur un bouton, quel qu'il soit, déclenchera une opération de traitement du clavier.

```
function initClavier()
{
    var div = $("#clavier");
    var html = "<table>";

    for (var ligne = 0; ligne < 4; ligne++)
    {
      html = html + "<tr>";
      for (var colonne = 0; colonne < 4; colonne++)
      {
        var value = (3 - ligne) * 4 + colonne;
        html = html + "<td><button class='chiffre base16";
        if (value < 10) html = html + " base10";
        if (value < 2) html = html + " base2";
        html = html + "' type='button' disabled>" +
    value.toString(16).toUpperCase() + "</button></td>";
      }
      html = html + "</tr>";
    }
    html = html + "</table>";
```

```
    div.html(html);
    $(".chiffre").click(function (event) {
        var elt = event.currentTarget;
        var touche = elt.innerText;
        var value = parseInt(touche, 16);
        var zoneSaisie = $("#nbr" + currentBase);
        var current = zoneSaisie.val();
        current = current.toString() + value.toString(16);
        zoneSaisie.val(current);
    });
}
```

Cette fonction sera appelée dès l'initialisation de l'application, dans la fonction `onDeviceReady`. La base de départ étant 10, les boutons des 10 premiers chiffres sont actifs :

```
initClavier();
$("base10").removeAttr("disabled");
```

La base courante est une variable globale, initialisée à 10 :

```
var currentBase = 10;
```

g) Développement : changement de base

Lors du changement de base (clic sur une zone de saisie) il faut effectuer les opérations suivantes :

- Déplacer le focus (déjà fait)
- Changer la base courante
- Remettre à zéro le contenu des zones de saisie

Il faut donc modifier la réponse à l'évènement `click` sur les zones de saisie pour ajouter les deux opérations :

```
$(".saisie").click(function (event) {
    putFocus(event.currentTarget);
    changeBase(event.currentTarget);
});
```

La remise à zéro des zones de saisie est simple à faire :

```
function raz() {
    $(".saisie").val("");
}
```

Le changement de base est un peu plus complexe : il faut en effet aller chercher le nombre (type `label`) contenu dans la `div` parent de la zone de saisie, et faire la mise à jour des attributs `disabled` des différents boutons :

```
function changeBase(elt) {
     currentBase =
     parseInt($(elt.parentElement).children("label")[0].inner
     Text);
     $(".chiffre").attr("disabled", "true");
     $(".base" + currentBase).removeAttr("disabled");
}
```

h) Développement : conversion

La conversion consiste à calculer, à chaque appui sur une touche, les autres bases à partir de la base courante.

La fonction de conversion sera appelée à la fin de la réponse à l'évènement `click` de chaque bouton-chiffre, en précisant la valeur à convertir :

```
conversion(zoneSaisie.val());
```

Cette fonction traduira en nombre (suivant la base courante) le paramètre, puis pour chaque base, convertira et affichera dans la zone de saisie *ad hoc* :

```
function conversion(val)
{
     var valeur = parseInt(val, currentBase);
     $(".saisie").each(function (index,zone) {
        var base =
     parseInt($(zone.parentElement).children("label")[0].inne
     rText);
        var conv = valeur.toString(base);
        zone.value = conv;
     });
}
```

Le test est possible : le calcul se fait bien mais, pour l'instant, le format n'est pas utilisé.

i) Développement : limitation du format

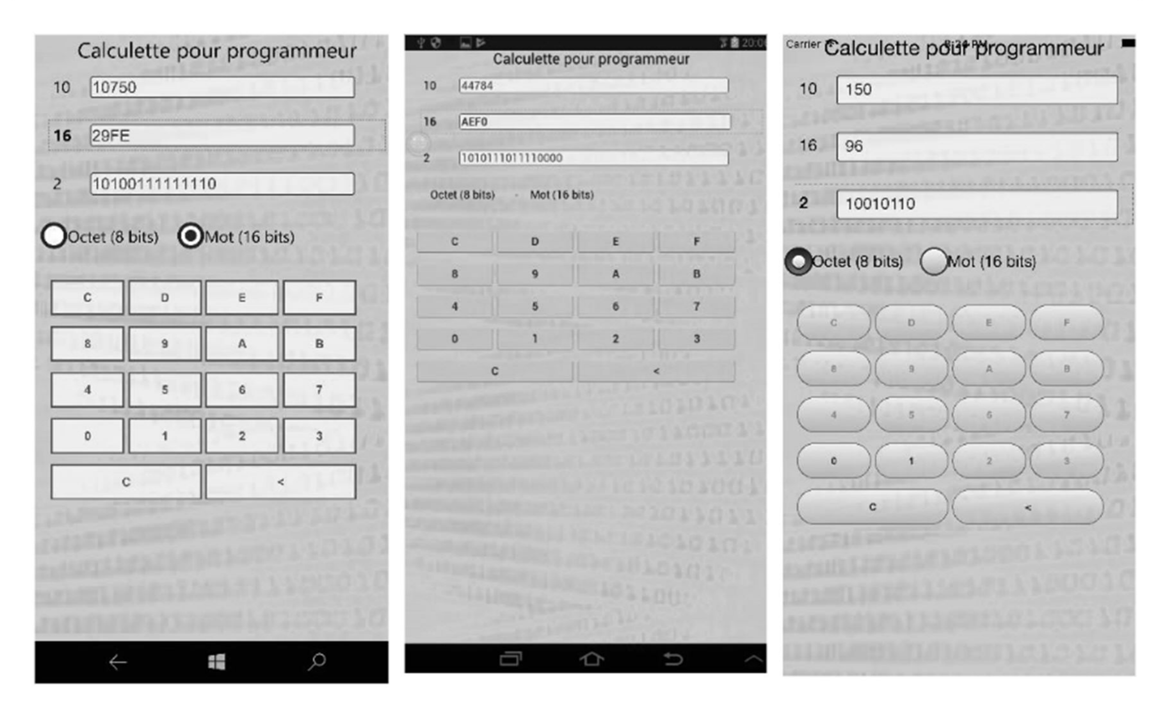
Le format d'un nombre (nombre de bits utilisés) limite le nombre de chiffres binaires possibles, ainsi que le nombre de chiffres dans les autres bases.

Le plus simple pour conserver cette limite est d'étudier, à chaque saisie de nombre (clic sur un bouton-chiffre) la valeur après rajout du chiffre : si celle-ci est supérieure à la valeur maximale du format, on ne peut rajouter ce chiffre. Il faut donc modifier la réponse à l'évènement `click` des boutons-chiffres :

```
$(".chiffre").click(function (event) {
     var elt = event.currentTarget;
     var touche = elt.innerText;
     var value = parseInt(touche, 16);
     var zoneSaisie = $("#nbr" + currentBase);
     var current = zoneSaisie.val();
     current = current.toString() + value.toString(16);
     value = parseInt(current, currentBase);
     if (value < (1 << bits)) { // on ne dépasse pas
        zoneSaisie.val(current);
        conversion(zoneSaisie.val());
     }
});
```

Il reste à rendre possible le changement de format en définissant la réponse à l'évènement `click` de chaque bouton radio (code à insérer dans l'opération `onDeviceReady`).

```
$(".format").click(function (event) {
     bits = parseInt(event.currentTarget.children[0].value);
     raz();
});
```



Capture 94 : calculatrice pour programmeur sous Windows, Android, iOS

j) Améliorations possibles

Cette application est volontairement très simple et limitée pour pouvoir être décrite totalement en peu de pages… La première version que j'ai déposée sur les différents magasins d'applications est celle décrite dans ces pages. Il est possible que je réalise des mises à jour de l'application, vous pouvez quant à vous essayer de l'améliorer en ajoutant des fonctionnalités, en modifiant l'IHM, etc…

Voici quelques idées de modifications intéressantes :

- Permettre d'utiliser des nombres de 32 bits (voire plus)
- Ajouter une touche pour supprimer le dernier chiffre saisi
- Améliorer l'aspect visuel, notamment par des images ou des couleurs,
- Gérer les nombres signés (complément à 2)

9. Carnet de santé pour animal

Difficulté : difficile

Plateformes ciblées : Windows 10, Android, iOS

Points techniques abordés : IHM, persistance des données, photo

Langages utilisés : JavaScript, HTML, CSS

Outils utilisés : Visual Studio, Cordova

Téléchargement de l'application :

- Pour Android : https://frama.link/wnTs7zth
- Pour Windows 10 : https://frama.link/xE12CboT
- Pour iOS : https://frama.link/W4UqRGMt

Téléchargement des codes sources : https://frama.link/9Y7mzjEN

a) Descriptif de l'application souhaitée

Afin de pouvoir suivre son animal familier, une application « carnet de santé » serait pratique.

L'application doit permettre de saisir :

- Les informations de son animal : nom, photo, date de naissance, espèce, race, sexe, numéro ID
- Les informations de son vétérinaire : nom, adresse, téléphone, mail
- Les interventions sur l'animal : vaccin, stérilisation, identification, etc… (nom, date intervention, commentaires)

b) Conception de l'application : design

L'application comprendra un écran d'accueil pour choisir les trois grandes parties : les informations sur l'animal, les informations sur le vétérinaire, et le « journal » lui-même.



L'écran d'accueil comprendra trois zones cliquables, avec des icônes gaies et simples et un texte rapide.



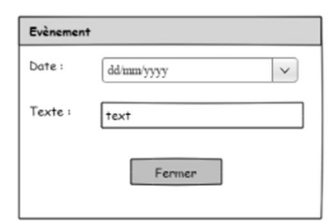
L'écran d'édition des informations de l'animal comprendra la photo de celui-ci, un bouton pour permettre de prendre cette même photo, des boutons radio pour choisir son genre, et des zones de saisies pour les informations diverses. L'écran servira à la fois à la visualisation et à la modification.



L'écran d'édition du vétérinaire comprendra des zones de saisie simples, une image pour égayer l'écran. Le clic sur le téléphone déclenchera l'appel, le clic sur le *mail* déclenchera la rédaction d'un *e-mail*.



L'écran « journal » listera les évènements de l'animal, en indiquant la date et un court texte. Un bouton permettra d'ajouter un nouvel évènement.



c) Conception de l'application : couche métier

Cette partie va contenir les classes nous permettant de modéliser le métier, c'est-à-dire le cœur de l'application, les notions qu'elle manipule.

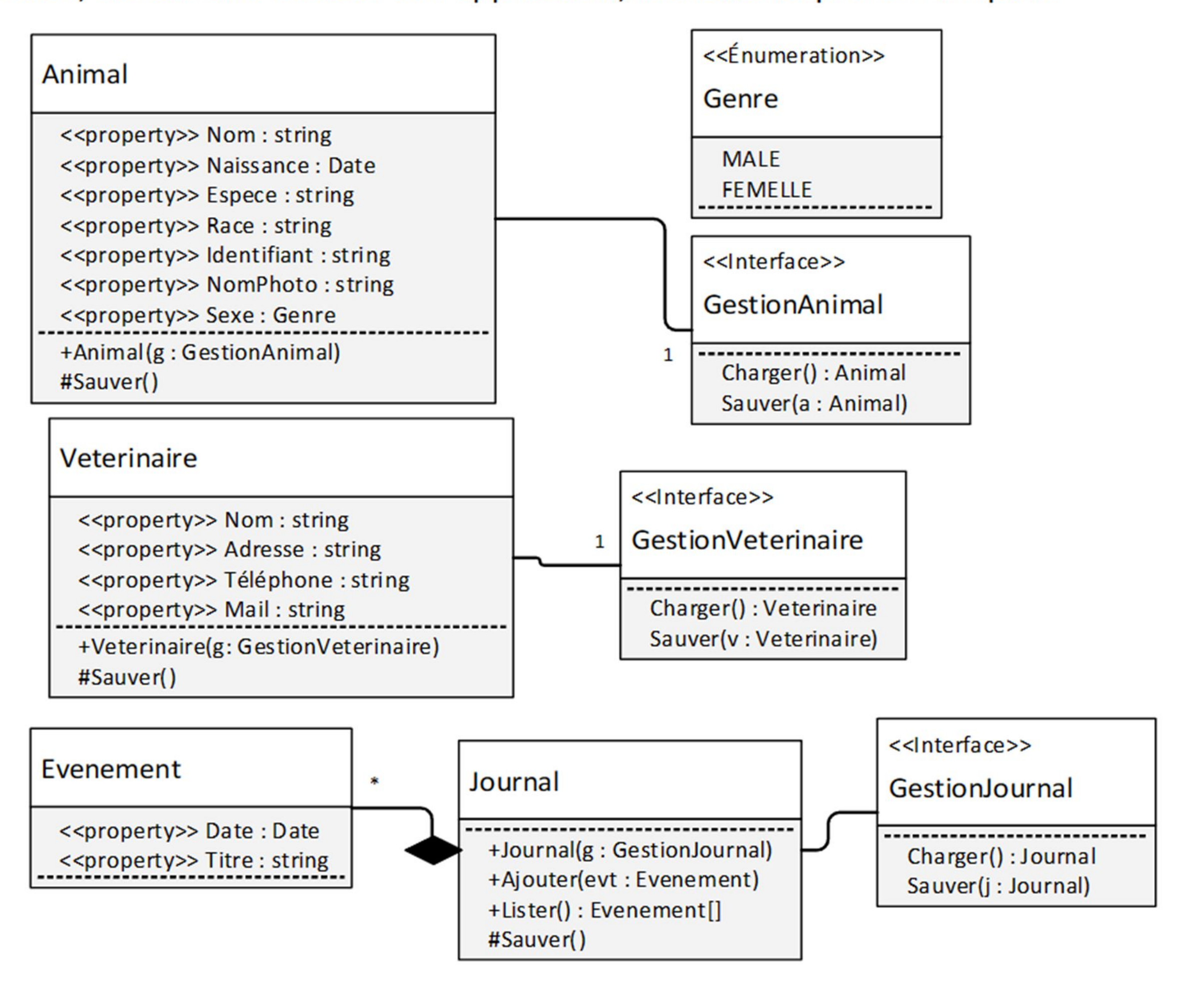


Diagramme 24 : classes métier atelier carnet de santé

Les classes stockant les données seront associées avec des interfaces de gestion créant une couche d'abstraction sur la sauvegarde des données. La méthode protégée `Sauver` sera appelée à chaque modification d'une

propriété, ce qui permettra de s'assurer que les données soient toujours à jour.

d) Conception de l'application : couche ihm

Notre IHM contient 4 écrans plus une boite de dialogue. Ces écrans sont dépendants de classes issues de la couche métier.

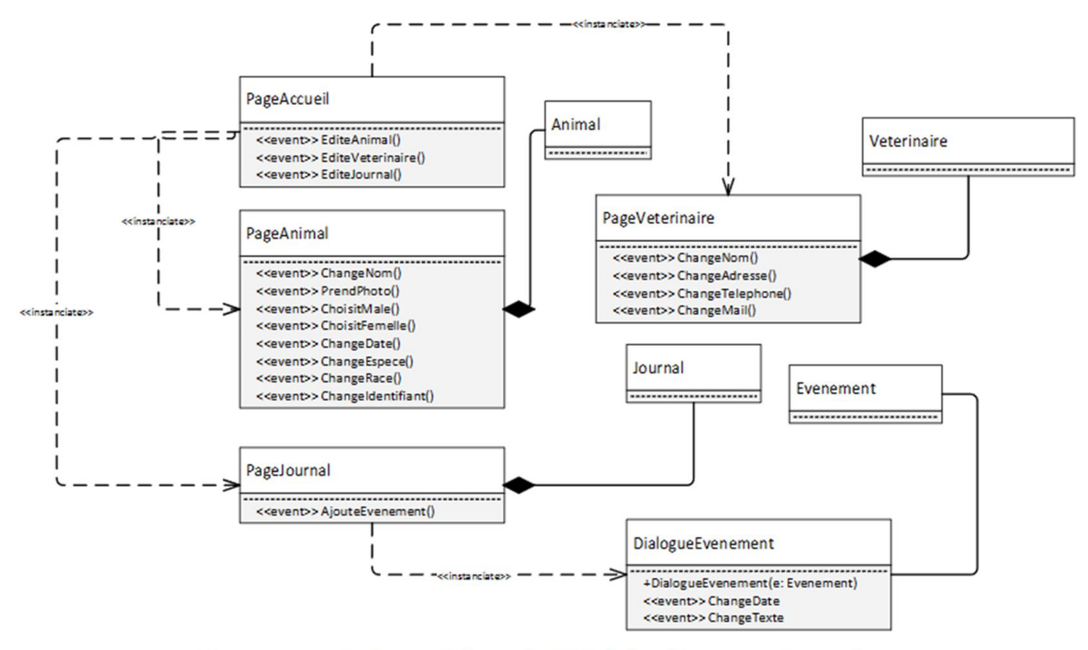


Diagramme 25 : classes de la partie IHM de l'atelier carnet de santé

e) Développement : page d'accueil

La page d'accueil n'utilise aucune classe de la couche métier, elle consiste simplement en un menu composé d'images et de textes et permettant de naviguer entre les différents écrans.

Pour implémenter la maquette de l'écran d'accueil en HTML, nous allons utiliser une division par entrée de menu. Cette division sera dotée d'une classe permettant de régler son apparence via la feuille de style CSS de la page. Chaque division contiendra un lien `<a>` contenant à la fois une image et du texte. Les images seront placées dans le dossier *ad hoc*.

```
<h1>Mon animal</h1>
<div class="item_menu">
    <a href="page_animal.html"><img
    src="images/chat.png"/>Infos</a>
</div>
```

```
<div class="item_menu">
     <a href="page_veto.html">Véto<img
     src="images/doctor.png"/></a>
</div>
<div class="item_menu">
     <a href="page_journal.html"><img
     src="images/journal.png"/>Bobos</a>
</div>
```

La division d'entrée de menu sera dotée d'un style permettant son alignement correct, un texte de grande taille entouré d'une marge :

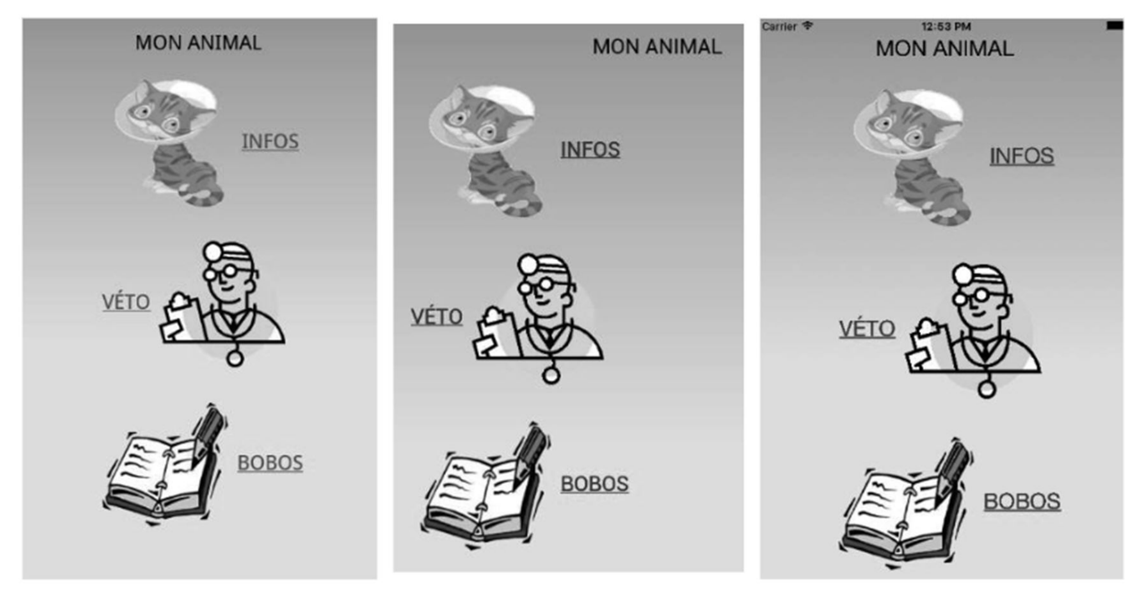
```
.item_menu {
    display:flex;
    font-size:x-large;
    margin:20px;
     width :100% ;
}
```

Le lien à l'intérieur de cette division devant être centré, ses marges sont définies sur `auto` :

```
.item_menu a{
    margin:auto;
}
```

Et enfin, l'image à l'intérieur de cette division doit être de taille fixe avec un alignement vertical centré :

```
.item_menu img {
    width:150px;
    height:150px;
    vertical-align:middle;
    margin:10px;
}
```



Capture 95 : écran d'accueil atelier carnet de santé pour Windows, Android, iOS

f) Développement : page vétérinaire

Cette page doit présenter les informations du vétérinaire. Nous allons créer une nouvelle page html (`page_veto.html`) qui sera la vue, un fichier lié pour le contrôleur (`ctrl_veto.js`), un fichier lié pour le modèle (`veterinaire.js`) ainsi qu'une classe pour gérer la persistance (`gere_veto.js`).

La page HTML à proprement parler est assez simple à écrire. Les champs de saisie seront tous identifiés pour pouvoir être manipulés dans le code.

```
<div class="screen">
     <div class="ligne">
       <input type="text" id="nom" />
     </div>
     <img src="images/doctor.png" />
     <div>
       <textarea id="adresse_veto" rows="4" ></textarea>
     </div>
     <div class="ligne">
       <img src="images/phone.png" id="phoneImg" />
         <input type="tel" id="phone" />
     </div>
       <div class="ligne">
       <img src="images/mail.png" id="mailImg" />
       <input type="email" id="mail" />
     </div>
</div>
```

Vous pouvez faire l'affichage que vous voulez en modifiant la CSS. Voici les valeurs que j'ai utilisées afin que l'aspect soit proche de la maquette.

Pour l'écran une certaine marge, avec une taille de police par défaut :

```
.screen{
    margin:10px;     padding:10px;     font-size:large;
}
```

La page étant plutôt disposée verticalement, les `div` auront une classe représentant une « ligne » de disposition :

```
.ligne{
    padding:10px;     width:80%;     margin:auto;
}
```

A l'intérieur de ces `div`, les images sont réduites et alignées sur le milieu du texte :

```
.ligne img{
    width:20%;
    height:auto;
    vertical-align:middle;
}
```

La zone pour saisir l'adresse doit occuper toute la largeur :

```
#adresse_veto{
    width:100%;
}
```

Les zones de saisie de texte et de date n'occupent qu'une partie de la « ligne » :

```
.ligne input[type="text"], input[type="date"] {
    border-radius: 4px;
    font-size:large;
    width:75%;
}
```

La zone pour saisir le nom utilise toute la largeur, le texte est en gras et plus gros :

```
#nom{
    width:100%;   font-size:x-large;   font-weight:bold;
}
```

Le modèle ne contient que des propriétés. En JavaScript, les attributs ne peuvent pas être privés (pas d'encapsulation au sens classique du terme) mais il est néanmoins préférable d'utiliser des opérations pour lire/écrire les attributs au lieu de les utiliser directement. Ici, par exemple, l'écriture d'une propriété déclenchera automatiquement la sauvegarde de l'objet.

```
class veterinaire {
    constructor(g) {
        this.gestion = g;
        this.nom = ""; this.adresse = "";
        this.telephone = "";
        this.mail = "";
    }

    sauver() {
        this.gestion.sauver(this);
    }

    getNom() { return this.nom; }
    setNom(value) { this.nom = value; this.sauver(); }
    getAdresse() { return this.adresse; }
    setAdresse(value) { this.adresse = value; this.sauver();
     }
    getTelephone() { return this.telephone; }
    setTelephone(value) { this.telephone = value;
     this.sauver(); }
    getMail() { return this.mail; }
    setMail(value) { this.mail = value; this.sauver(); }
}
```

Pour la persistance, nous utiliserons simplement une sérialisation JSON et le *local storage* du navigateur (voir page VII.1.e).

```
class gere_veterinaire {
    charger() {
        var veto = null;
        var storage = window.localStorage;
        var txt = storage.getItem("veterinaire");
        if (txt == null || txt == "")
        {
            veto = new veterinaire(this);
        }
        else {
            var tmp = JSON.parse(txt);
            veto = Object.assign(new veterinaire(this), tmp);
            veto.gestion = this;
        }
```

```
        return veto;
    }

    sauver(veto) {
        var storage = window.localStorage;
        var txt = JSON.stringify(veto);
        storage.setItem("veterinaire", txt);
    }
}
```

Le contrôleur est chargé à l'ouverture de la page. Il commence par initialiser le gestionnaire et charge l'objet représentant le vétérinaire :

```
window.onload = () => {
    var gestion = new gere_veterinaire();
    var veto = gestion.charger();
```

Les champs de la vue sont initialisés à l'aide des propriétés du modèle. J'ai eu recours à *JQuery* pour simplifier le code, mais ce n'est pas obligatoire.

```
$("#nom").val(veto.getNom());
$("#adresse").val(veto.getAdresse());
$("#phone").val(veto.getTelephone());
$("#mail").val(veto.getMail());
```

Pour modifier le modèle directement, nous allons utiliser les évènements `focusout` (déclenchés quand l'utilisateur « sort » d'une zone de saisie) pour écrire dans le modèle les nouvelles valeurs :

```
$("#nom").on("focusout", () => {
     veto.setNom($("#nom").val()); });
$("#adresse").on("focusout", () => {
     veto.setAdresse($("#adresse").val()); });
$("#phone").on("focusout", () => {
     veto.setTelephone($("#phone").val()); });
$("#mail").on("focusout", () => {
     veto.setMail($("#mail").val()); });
```

Et enfin, pour lancer l'appel téléphonique ainsi que l'e-mail :

```
$("#phoneImg").on("click", function () {
     window.location.href = "tel:" + veto.getTelephone();
});
$("#mailImg").on("click", function () {
     window.location.href = "mailto:" + veto.getMail();
});
```



Capture 96 : page vétérinaire, application carnet de santé (Windows, Android, iOS)

g) Développement : page animal

Cette page doit présenter les informations d'un animal. Elle est donc reliée à la couche métier. Nous allons ainsi obtenir une nouvelle page html (`page_animal.html`) qui sera la vue, ainsi qu'un fichier JavaScript lié (`ctrl_animal.js`) qui servira de contrôleur, un fichier JavaScript (`animal.js`) qui servira de modèle.

La page HTML sera découpée avec un certain nombre de divisions, et une table pour organiser de manière plus ordonnée les champs :

```
<div class="screen">
        <div class="ligne">
            <input type="text" id="nom" />
        </div>
        <div id="photos" class="ligne">
            <img id="photo" />
            <img id="camera" src="images/camera.png" />
        </div>
        <div id="genre" class="ligne">
            <input type="radio" name="genre" id="male" />
            <label for="male">Mâle</label>
            <input type="radio" name="genre" id="femelle" />
            <label for="femelle">Femelle</label>
        </div>
        <table id="donnes_animal">
            <tr><td><label for="naissance">Né le
    :</label></td>
                <td><input type="date" id="naissance"
    /></td></tr>
            <tr><td><label for="espece">Espèce :</label></td>
                <td><input type="text" id="espece"
```

```
    /></td></tr>
                <tr><td><label for="race">Race :</label></td>
                    <td><input type="text" id="race" /></td></tr>
                <tr><td><label for="id">Identifiant
    :</label></td>
                    <td><input type="text" id="id" /></td></tr>
        </table>
</div>
```

La classe `animal` est assez simple, elle ne contient que des propriétés ; dans chaque mutateur (la fonction permettant d'écrire la propriété), nous forcerons la sauvegarde du modèle en utilisant la classe associée (le gestionnaire).

```
class animal {
    constructor(g) {
        this.nom = "";
        this.espece = "";
        this.race = "";
        this.naissance = "";
        this.identifiant = "";
        this.photo = "";
        this.genre = "";
        this.gestion = g;
    }

    getNom() { return this.nom; }
    setNom(value) { this.nom = value; this.sauver(); }

    getEspece() { return this.espece; }
    setEspece(value) { this.espece = value; this.sauver(); }

    getRace() { return this.race; }
    setRace(value) { this.race = value; this.sauver(); }

    getNaissance() { return this.naissance; }
    setNaissance(value) { this.naissance = value;
     this.sauver(); }

    getIdentifiant() { return this.identifiant; }
    setIdentifiant(value) { this.identifiant = value;
     this.sauver(); }

    getPhoto() { return this.photo; }
    setPhoto(value) { this.photo = value; this.sauver(); }

    getGenre() { return this.genre; }
    setGenre(value) { this.genre = value; this.sauver(); }
```

```
    sauver() {
        this.gestion.sauver(this);
    }
}
```

La gestion de la persistance des données est délocalisée dans une classe qui utilisera le stockage local et la sérialisation JSON.

```
class gere_animal {
    charger()
    {
        var storage = window.localStorage;
        var txt = storage.getItem("animal");
        var a = null;
        if (txt != null && txt != "")
        {
            var tmp = JSON.parse(txt);
            a = Object.assign(new animal(this), tmp);
            a.gestion = this;
        }
        else  // première exécution
            a = new animal(this);
        return a;
    }

    sauver(a)     {
        console.log(a);
        var storage = window.localStorage;
        var txt = JSON.stringify(a);
        storage.setItem("animal", txt);
    }
}
```

Dans le contrôleur, nous allons commencer par réagir à la fin du chargement de la page (inutile de faire quoi que ce soit avant), créer le gestionnaire et charger l'objet représentant l'animal :

```
window.onload = function () {
    var gestion = new gere_animal();
    var a = gestion.charger();
```

Puis il faut récupérer les propriétés de l'objet animal chargé pour les afficher dans les différents champs.

```
   $("#nom").val(a.getNom());
    $("#naissance").val(a.getNaissance());
```

```
    $("#espece").val(a.getEspece());
    $("#race").val(a.getRace());
    $("#id").val(a.getIdentifiant());
    var genre = a.getGenre();
    if (genre == "male")
        $("#male").prop("checked", true);
    else if (genre == "femelle")
        $("#femelle").prop("checked", true);
    chargePhotoBase64(a.getPhoto());
```

Il faut ensuite détecter les modifications des zones de saisies (`input`) réalisées par l'utilisateur en utilisant l'évènement `focusout` :

```
$("#nom").on("focusout", () => { a.setNom($("#nom").val());
     });
$("#naissance").on("focusout", () => {
        a.setNaissance($("#date").val()); });
$("#espece").on("focusout", () => {
        a.setEspece($("#espece").val()); })
$("#race").on("focusout", () => {
     a.setRace($("#race").val());});
$("#id").on("focusout", () => {
     a.setIdentifiant($("#id").val()); });
```

Pour les boutons radio, l'évènement `click` est plus pratique :

```
$("#male").on("click", () => { a.setGenre("male"); });
$("#femelle").on("click", () => { a.setGenre("femelle");
```

Enfin, pour la photo, nous utilisons le *plugin* déjà évoqué page 194 :

```
$("#camera").on("click",() => {
        navigator.camera.getPicture(
            (imageData) => {
                chargePhotoBase64(imageData);
                a.setPhoto(imageData);
            },
            () => {
                console.log("Capture de l'image ratée");
            },
            { quality: 50, sourceType:
     Camera.PictureSourceType.CAMERA, destinationType:
     Camera.DestinationType.DATA_URL, cameraDirection:
     Camera.Direction.FRONT });
    });
```

La dernière fonction affiche tout simplement l'image (codée en base64) :

```
function chargePhotoBase64(data)
{
    $("#photo").prop("src", "data:image/jpg;base64," + data);
}
```







Capture 97 : atelier carnet de santé, page animal, sous Windows, Android, iOS

h) Développement : page journal

Nous allons réaliser pour le journal une page très simple. Le haut de la page comprendra le bouton de retour, le titre, une image :

```
<div class="back">
<button type="button" id="back"></button></div>
<h1>Bobos</h1><img src="images/journal.png" id="journal" />
```

Nous utiliserons ensuite une liste pour afficher les entrées du journal, ainsi qu'un bouton pour ajouter une entrée :

```
<list id="events"></list>
<button id="ajouter">Ajouter</button>
```

Le style de la liste est le suivant :

```
#events{
    display:block;
    padding:10px;
    width:100%; height:60%;
    min-height:200px;
    margin: 0 10px 10px 10px;
    border: 1px solid #AAA;
    box-sizing: border-box;
    overflow-y: auto;
}
```

```css
#ajouter, #fermer {
    background-color: #333;
    border-radius: 5px;
    border: 1px solid #222;
    color: #FFF;
    cursor: pointer;
    box-shadow: 0px 0px 4px rgba(0, 0, 0, 0.8);
    font-size:x-large;
    float:right;
}
```

Afin de saisir une entrée de journal, nous utiliserons une autre page HTML comprenant une zone de saisie de date, une zone de saisie de texte ainsi qu'un bouton :

```html
<div class="ligne">
    <label for="date">Date :</label>
    <input type="date" id="date" />
</div>
<div class="ligne">
    <label for="titre">Titre :</label>
    <input type="text" id="titre" />
</div>
<button id="fermer">Fermer</button>
```

Voici la classe `journal` :

```javascript
class journal {
    constructor(g) {
        this.gestion = g;
        this.events = [];
    }

    ajouter(evt) {
        this.events.push(evt);
        this.sauver();
    }

    lister(evt) {
        return this.events;
    }

    sauver() {
        this.gestion.sauver(this);
    }
}
```

Son gestionnaire est similaire à ceux du vétérinaire ou de l'animal :

```
charger() {
        var log = null;
        var txt = window.localStorage.getItem("journal");
        if (txt == null || txt=="")    {
            log = new journal(this);
        }
        else{
            var tmp = JSON.parse(txt);
            log = Object.assign(new journal(this), tmp);
            log.gestion = this;
        }
        return log;
    }
    sauver(journal) {
        var txt = JSON.stringify(journal);
        window.localStorage.setItem("journal", txt);
    }
```

La classe `evenement` est basique :

```
class evenement {
    constructor(date,titre) {
        this.date = date
        this.titre = titre;
    }

}
```

Dans le contrôleur lié au journal, commençons par initialiser les objets :

```
var gestion = new gere_journal();
var journal = gestion.charger();
var events = journal.lister();
```

Il faut afficher dans la liste les évènements en récupérant leurs données :

```
let html = "";
events.forEach( function(item, index)  {
     html += '<div class="event">';
     let tmp = new Date(item.date);
     let str = tmp.toLocaleDateString();
     html += '<span class="date">' + str + '</span><span
     class="titre">' + item.titre + "</span>";
     html += '</div>';
});
$("#events").html(html);
```

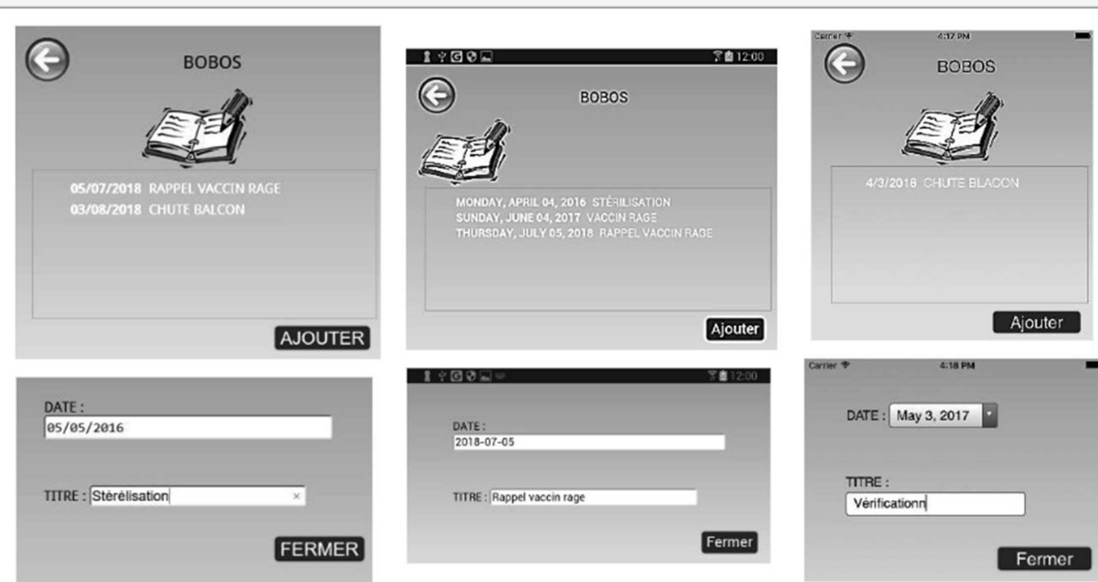
Et enfin, lier au clic sur le bouton `ajouter` le changement de page :

```
    $("#ajouter").on("click", function() {
        window.location.href = "page_event.html";
    });
```

Pour finir, le contrôleur pour l'évènement :

```
window.onload = function () {
    ctrl_event();
}
function ctrl_event() {
    var gere = new gere_journal();
    var journal = gere.charger();

    $("#fermer").on("click", function() {
        var evt = new evenement($("#date").val(),
     $("#titre").val());
        journal.ajouter(evt);
        window.history.back();
    });
}
```



Capture 98 : page journal avec Windows, Android, iOS

i) Améliorations possibles

Cette application est volontairement très simple et limitée pour pouvoir être décrite totalement en peu de pages… La première version que j'ai déposée sur les différents magasins d'applications est celle décrite dans ces pages. Il est possible que je réalise des mises à jour de l'application, vous

pouvez quant à vous essayer de l'améliorer en ajoutant des fonctionnalités, en modifiant l'IHM, etc…

Voici quelques idées de modifications intéressantes :

- Améliorer l'IHM par une application mono-page (utilisation d'une page à onglets par exemple, la bibliothèque *JQueryUI* peut être utilisée pour simplifier).
- Permettre de fixer une date approximative de prochain RV (pour vaccin, rappel, etc.) avec le vétérinaire qui créera une alarme pour faire penser à prendre un RV.
- Plus d'informations dans le journal, avec possibilité de recherche.

10. Météo

Difficulté : difficile

Plateformes ciblées : Windows 10, Android, iOS

Points techniques abordés : IHM, service web, géolocalisation

Langages utilisés : C++

Outils utilisés : Qt Creator

Téléchargement de l'application :

- Pour Android : https://frama.link/oGgh2Qz3
- Pour Windows 10 : https://frama.link/_04UubbH
- Pour iOS : https://frama.link/pDBrW-GZ

Téléchargement des codes sources : https://frama.link/hXoayDyU

a) Descriptif de l'application souhaitée

Nous allons créer une petite application pour connaître la météo, en utilisant un service web. L'application permettra donc d'obtenir la météo du lieu courant (en utilisant la géolocalisation) ou d'une ville quelconque.

Nous utiliserons le service gratuit `OpenWeatherMap` (28) qui nécessite néanmoins une inscription sur leur site. Consultez bien leurs conditions pour connaître les limitations du service gratuit.

b) Conception de l'application : *design*

Une seule fenêtre pour l'application, permettant d'afficher la météo simplement et de passer aux jours suivants.



L'écran contiendra un bouton avec logo de géolocalisation qui servira à demander la météo du lieu précis où on se trouve, une zone de saisie pour entrer un nom de ville avec un bouton pour lancer la recherche météo dans cette ville, des boutons sur les côtés pour passer aux horaires suivants/précédents et bien entendu, les différentes informations météo sous forme de texte simple. Une petite image représentant le type de temps est aussi utile !

c) Conception de l'application : couche métier

La couche contient tout d'abord les classes nécessaires pour stocker les données météo. Une interface pour fournir ces informations fait également partie de la couche, ainsi qu'une classe regroupant les fonctionnalités nécessaires à la couche de présentation. Il faut également une interface pour le fournisseur de localisation :

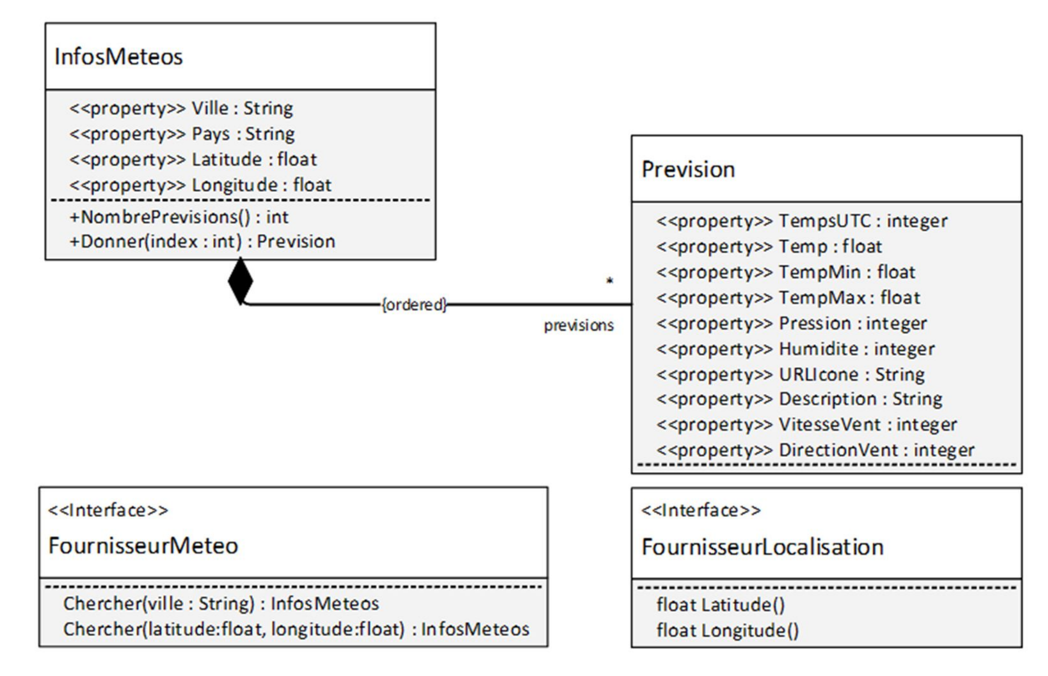


Diagramme 26 : diagramme de classes couche métier appli météo

d) Conception de l'application : couche IHM

La couche de présentation contient une classe représentant l'écran unique de l'application, et regroupant les opérations nécessaires pour répondre aux évènements utilisateur :

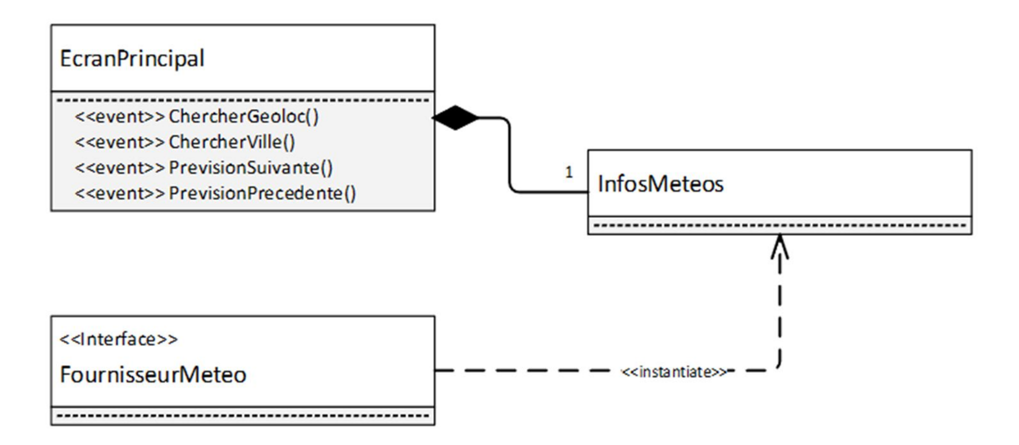


Diagramme 27 : couche IHM de l'application météo

e) Conception de l'application : couche réseau

Cette couche va contenir la classe qui va s'occuper de la connexion au service web. Nous utiliserons un *proxy* pour pouvoir éventuellement placer en cache des données.

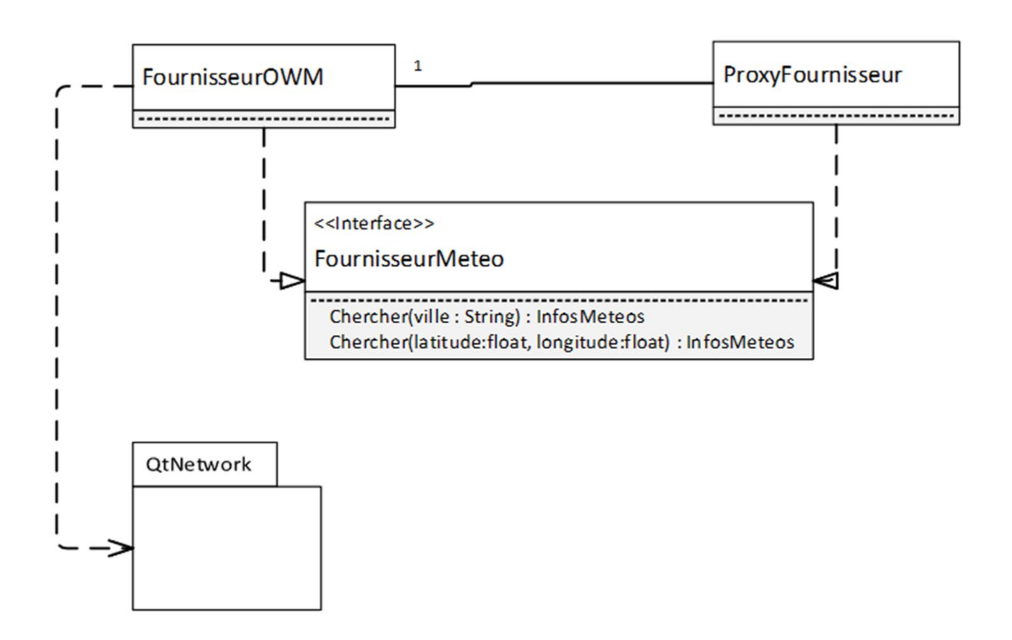


Diagramme 28 : couche réseau de l'application Météo

f) Développement de l'application : couche métier

Commençons par la classe `Prevision` qui ne contient que des propriétés ; dans le code suivant, les mutateurs sont volontairement simples, voici donc juste le fichier `prevision.h` :

```cpp
class Prevision
{
    int tempsUTC;
    int temp;
    float tempMin;
    float tempMax;
    int pression;
    int humidite;
    std::string urlIcone;
    std::string description;
    int vitesseVent;
    std::string directionVent;
    float pluieMM;
    float neigeMM;
public:
    int getTempsUTC() const;
    void setTempsUTC(int value);
    int getTemp() const;
    void setTemp(int value);
    float getTempMin() const;
    void setTempMin(float value);
    float getTempMax() const;
    void setTempMax(float value);
    int getPression() const;
    void setPression(int value);
    int getHumidite() const;
    void setHumidite(int value);
    std::string getUrlIcone() const;
    void setUrlIcone(const std::string &value);
    std::string getDescription() const;
    void setDescription(const std::string &value);
    int getVitesseVent() const;
    void setVitesseVent(int value);
    std::string getDirectionVent() const;
    void setDirectionVent(float value);
    float getPluieMM() const;
    void setPluieMM(float value);
    float getNeigeMM() const;
    void setNeigeMM(float value);
};
```

La classe `InfosMeteo` contient tout d'abord un tableau de prévisions, un ensemble de propriétés (attribut, accesseur et mutateur), des opérations pour accéder aux données :

```cpp
class InfosMeteo
{
    std::vector<Prevision> previsions;
    std::string ville;
    std::string pays;
    float latitude;
    float longitude;
public:
    std::string getVille() const;
    void setVille(const std::string &value);
    std::string getPays() const;
    void setPays(const std::string &value);
    float getLatitude() const;
    void setLatitude(float value);
    float getLongitude() const;
    void setLongitude(float value);
    int nombrePrevisions() const {return previsions.size();}
    Prevision donner(int index) const {return
     previsions.at(index);}
    void ajouter(Prevision p){previsions.push_back(p) ;}
     std::string getLatitudeStr() const ;
     std::string getLongitudeStr() const ;
};
```

Le code des accesseurs/mutateurs n'est pas donné mais il est très simple, voici celui pour la ville :

```cpp
std::string InfosMeteo::getVille() const
{
    return ville;
}

void InfosMeteo::setVille(const std::string &value)
{
    ville = value;
}
```

Les deux opérations servant à fournir latitude et longitude sous une forme de chaîne de caractères plus lisible :

```cpp
std::string InfosMeteo::getLatitudeStr() const
{
```

```
    std::ostringstream flux;
    float lat = latitude;
    if(latitude>0)
        flux << "N ";
    else if(latitude<0)
    {
        flux << "S ";
        lat = -latitude;
    }
    int deg = (int)lat;
    float min = (lat-deg)*60;
    flux << deg<<"."<<min;
    return flux.str();
}

std::string InfosMeteo::getLongitudeStr() const
{
    std::ostringstream flux;
    float longi=longitude;
    if(longitude>0)
        flux << "W ";
    else if(longitude<0)
    {
        flux << "E ";
        longi = -longitude;
    }
    int deg = (int)longi;
    float min = (longi-deg)*60;
    flux << deg<<"."<<min;
    return flux.str();
}
```

Il ne reste dans la couche que la définition de l'interface :

```
class FournisseurMeteo{
public:
    virtual ~FournisseurMeteo(){}
    virtual InfosMeteo Chercher(std::string ville) const = 0;
    virtual InfosMeteo Chercher(float longitude, float
     latitude) const = 0;
};
```

Pour les futurs tests de la couche, un fournisseur factice est très pratique :

```
class FournisseurFactice : public FournisseurMeteo
{
public:
    InfosMeteo Chercher(std::string ville) const;
```

```
    InfosMeteo Chercher(float longitude, float latitude)
     const;
};

InfosMeteo FournisseurFactice::Chercher(float longitude,
     float latitude) const
{
    return Chercher("");
}

InfosMeteo FournisseurFactice::Chercher(std::string ville)
     const
{
    InfosMeteo infos;
    infos.setLatitude(48.86);
    infos.setLongitude(2.35);
    infos.setPays("France");
    infos.setVille("Paris");

    Prevision p;
    p.setDescription("Plein soleil");
    p.setDirectionVent(60);
    p.setHumidite(40);
    p.setPression(1024);
    p.setTemp(25);
    p.setTempMax(28);
    p.setTempMin(12);
    p.setTempsUTC(1534001195); // 11/8/2018 17:26:35
    p.setUrlIcone("01d");
    p.setVitesseVent(3);
     p.setPluieMM(0);
    p.setNeigeMM(0);

    infos.ajouter(p);
    p.setTempsUTC(1534011995); // 3h après
    p.setDescription("Nuageux");
    p.setHumidite(70);
    p.setPression(1000);
    p.setUrlIcone("02d");
     p.setPluieMM(0.3) ;
    infos.ajouter(p);

    return infos;
}
```

L'interface pour le fournisseur de localisation est la suivante :

```
class FournisseurLocalisation
{
public:
    virtual ~FournisseurLocalisation(){}
    virtual float Longitude() const = 0;
    virtual float Latitude() const = 0;
};
```

Là également, un fournisseur factice sera pratique pour tester :

```
class FournisseurLocalisationFactice : public
     FournisseurLocalisation
{
public:
    float Longitude() const {return 48.356;}
    float Latitude() const {return 2.385;}
} ;
```

g) Développement : accès http

Nous aurons besoin d'un accès *http* dans deux cas :

- Demander les informations météo auprès du service web
- Récupérer l'image correspondant au temps

Il nous suffit d'écrire une classe `HttpRequest` qui s'occupera de réaliser cette tâche, de manière asynchrone bien sûr.

Cette classe doit hériter de `QObject` :

```
class HttpRequest : public QObject
{
    Q_OBJECT
```

Elle va fournir une opération fournissant un `QByteArray` (tableau d'octets) à partir d'une URL :

```
public:
    QByteArray get(QString txt);
```

Il faut également un *slot* dans cette classe pour récupérer le retour de la requête :

```
private slots:
    void requeteOk(QNetworkReply *reply);
```

Deux attributs seront nécessaires pour transférer des informations entre les deux opérations : un booléen qui indique que la réponse est arrivée, et un tableau d'octets pour stocker celle-ci :

```
private:
    bool ok=false;
    QByteArray data;
```

Pour le *slot* qui récupère la réponse, il suffit de lire celle-ci dans le tableau d'octets :

```
void HttpRequest::requeteOk(QNetworkReply *reply)
{
    ok=true;
    data = reply->readAll();
}
```

Et enfin, l'opération qui exécute la requête *http* est la suivante :

```
QByteArray HttpRequest::get(QString txt)
{
    QNetworkAccessManager manager;
    QUrl url(txt);
    QNetworkRequest req(url);
    connect(&manager,SIGNAL(finished(QNetworkReply*)),this,
     SLOT(requeteOk(QNetworkReply*)));
    ok=false;
    manager.get(req);
    while(!ok)
        QCoreApplication::processEvents(); // "passe la main"
    return data;
}
```

h) Développement : couche IHM

Créons un fichier de ressources Qt pour placer les différentes images utiles pour l'interface graphique, comme vu page 141.

Afin de disposer correctement les différents contrôles pour respecter la maquette, il faut les positionner dans des *layouts* ou directement dans des *widgets*. L'écran complet aura une disposition verticale.

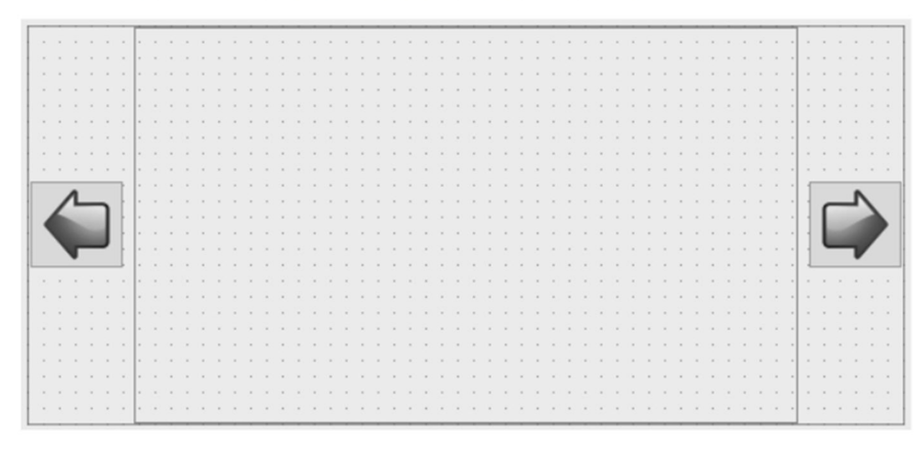
En haut de l'écran, un *widget* avec un *layout* horizontal contenant une image, une zone de saisie ville et un bouton :



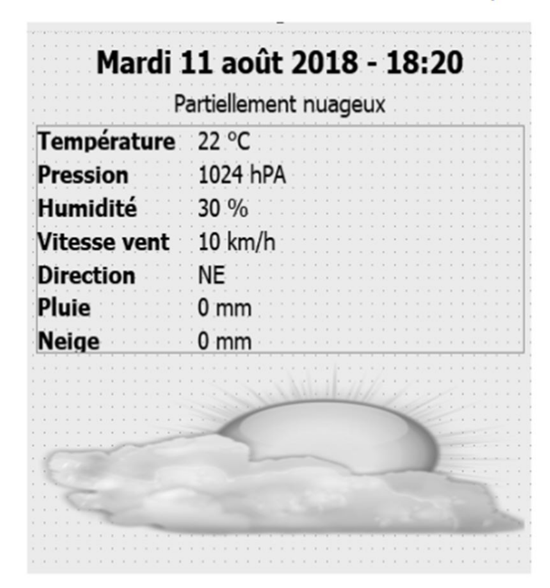
Le *widget* suivant est aussi horizontal et contiendra la latitude et la longitude.



Le *widget* suivant remplira le reste de l'écran, il sera de type horizontal et contiendra un bouton, un *widget* et un autre bouton :



A l'intérieur du *widget* central, la disposition sera verticale. On trouvera bien entendu les contrôles texte nécessaire (tous de type *label*).



Evidemment, l'application a besoin d'un fournisseur pour la météo, donc un objet de type `FournisseurMeteo` comme attribut, un objet de type

`FournisseurLocalisation`, ainsi que le lien sur la couche métier et un entier indiquant la prévision courante :

```
private:
    FournisseurMeteo* fournisseur;
     InfosMeteos infos ;
     int courante = -1 ;
    FournisseurLocalisation* localisation;
```

Ces fournisseurs seront bien entendu initialisés dans le constructeur de la fenêtre :

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    fournisseur = new FournisseurFactice();
    localisation = new FournisseurLocalisationFactice();
}
```

Et la mémoire sera libérée dans le destructeur :

```
MainWindow::~MainWindow() {
    delete fournisseur;
    delete localisation;
    delete ui;
}
```

Il faut créer une opération privée qui affiche les informations météo :

```
    void afficher();
```

Le clic sur le bouton pour demander la géolocalisation va donc effectuer le travail :

```
void MainWindow::chercherLongLat()
{
    float latitude = localisation->Latitude();
    float longitude = localisation->Longitude();
    infos = fournisseur->Chercher(longitude,latitude);
    courante = 0 ;
    afficher();
}
```

Le code pour récupérer la météo à partir du nom d'une ville est le suivant :

```
void MainWindow::chercherVille()
{
    std::string ville = ui->ville->text().toStdString();
    if(!ville.empty())
    {
        infos = fournisseur->Chercher(ville);
        courante = 0;
        afficher();
    }
}
```

Pour changer de prévision, le code de réponse aux clics sur les boutons est le suivant :

```
void MainWindow::precedent()
{
    if(courante>0)
        --courante;
    afficher();
}

void MainWindow::suivant()
{
    if(courante<infos.nombrePrevisions()-1)
        ++courante;
    afficher();
}
```

Et enfin, le code nécessaire à l'affichage de la prévision : il nécessite de nombreuses conversions, notre couche métier utilisant `std::string` pour les chaînes et les contrôles Qt la classe `QString`. Au départ, il convient d'afficher les informations générales :

```
void MainWindow::afficher()
{
    ui->ville-
     >setText(QString::fromStdString(infos.getVille()));
    ui->latitude->setText(
     QString::fromStdString(infos.getLatitudeStr()));
    ui->longitude->setText(
     QString::fromStdString(infos.getLongitudeStr()));
    ui->ville-
     >setText(QString::fromStdString(infos.getVille()));
```

Puis de récupérer la prévision courante :

```
    Prevision p = infos.donner(courante);
```

La date de la prévision étant donnée en temps Unix[1], nous allons utiliser la classe `QDateTime` du *framework* Qt, capable d'utiliser le temps Unix et de s'afficher d'une manière dépendante du pays courant :

```
QDateTime dt = QDateTime::fromTime_t(p.getTempsUTC());
ui->date->setText(dt.toString("dddd dd MMMM yyyy - HH:mm"));
```

Les informations suivantes nécessitent des simples conversions en chaînes, et l'ajout de l'unité pour faciliter la lecture :

```
ui->description->
     setText(QString::fromStdString(p.getDescription()));
ui->direction->
     setText(QString::fromStdString(p.getDirectionVent()));
ui->vitesseVent->setText(QString::number(p.getVitesseVent())+
     "km/h");
ui->humidite->setText(QString::number(p.getHumidite()) + "
     %");
ui->pression->setText(QString::number(p.getPression())+ "
     hPA");
ui->temperature->setText(QString::number(p.getTemp()) + "
     °C");
ui->pluie->setText(QString::number(p.getPluieMM())+" mm");
ui->neige->setText(QString::number(p.getNeigeMM())+" mm");
```

Il ne reste qu'à afficher l'icône représentant le temps. Cette image est fournie par le site *openWeatherMap*, il nous suffit donc de faire une nouvelle requête *http* et de placer l'image dans le contrôle *ad hoc*. L'image est agrandie (ici à 200 pixels de largeur) car l'icône originale est très petite :

```
QString url = "http://openweathermap.org/img/w/" +
  QString::fromStdString(p.getUrlIcone()) + ".png";
HttpRequest req;
QByteArray data = req.get(url);
QPixmap bmp;
bmp.loadFromData(data);
ui->icone->
  setPixmap(bmp.scaledToWidth(200,Qt::SmoothTransformation));
```

[1] Norme de représentation de la date, sous forme du nombre de secondes écoulées depuis une date de référence (en général le 1er janvier 1970 à minuit), ce qui permet avec un seul entier d'avoir la date et l'heure précise à la seconde.

i) Développement : couche fournisseur météo

On accède au service web à l'aide de la classe développée au XII.10.g). L'URL de base du service est :

`api.openweathermap.org/data/2.5/forecast`.

Le retour est en JSON, comme décrit au (28).

Il nous faut donc une classe qui implémente l'interface *ad hoc* :

```cpp
class OpenWeatherMap : public FournisseurMeteo
{
```

La clé d'API nécessaire pour utiliser le service web (à obtenir en s'inscrivant sur le site *OpenWeatherMap*) ainsi que l'URL de base du service seront stockées dans les attributs de cette classe :

```cpp
private:
    QString urlBase;
    QString apiKey;
```

Les attributs seront initialisés dans le constructeur (utilisez votre propre clé à la place des symboles `XXX`) :

```cpp
OpenWeatherMap::OpenWeatherMap()
    : apiKey("XXXXXXXXXXXXXXXXXXXXXXXXXXXXX"),
     urlBase("api.openweathermap.org/data/2.5/forecast?lang=f
     r")
{
}
```

Les deux opérations de l'interface utiliseront toutes deux la classe `HttpRequest` ainsi qu'une opération privée de la classe pour analyser le retour :

```cpp
InfosMeteo OpenWeatherMap::Chercher(float longitude, float
     latitude) const
{
    QString url =
     urlBase+"&APPID="+apiKey+"&lat="+QString::number(latitud
     e)+"&lon="+QString::number(longitude);
    HttpRequest req;
    QByteArray data = req.get(url);
    return Lire(data);
}
```

```
InfosMeteo OpenWeatherMap::Chercher(std::string ville) const
{
    QString url =
     urlBase+"&APPID="+apiKey+"&q="+QString::fromStdString(vi
     lle);
    HttpRequest req;
    QByteArray data = req.get(url);
    return Lire(data);
}
```

j) Développement : fournisseur de position

Nous allons créer une nouvelle classe pour le fournisseur de position. Celle-ci doit implémenter notre interface, mais également hériter de `QObject` pour pouvoir récupérer les signaux du GPS.

```
class FournisseurGPS : public QObject, public
     FournisseurLocalisation
{
    Q_OBJECT
public:
    FournisseurGPS();
    float Latitude() const;
    float Longitude() const;
private slots:
    void nouvellePosition(const QGeoPositionInfo& info);
```

En attributs, cette classe contiendra un lien vers un objet de type `QGeoPositionInfoSource`, deux flottants pour stocker latitude et longitude, un booléen pour l'état de la mesure, et une valeur de type `QDateTime` pour la date précise de la dernière mesure de la position (pour éviter de mesurer en permanence).

```
private:
    QGeoPositionInfoSource* source;
    float latitude=0;
    float longitude=0;
    mutable bool ok=false;
    QDateTime tempsMesure;
```

Et deux opérations privées : une pour prendre la mesure de la localisation, une pour savoir si la mesure précédente est encore valide :

```
    bool mesureOk() const;
    void mesure() const;
```

Le constructeur initialisera le lien vers le `QGeoPositionInfoSource` et connectera le *slot* avec le signal :

```
FournisseurGPS::FournisseurGPS() {
    source =
     QGeoPositionInfoSource::createDefaultSource(this);
    if(source) {
        connect(source,
     SIGNAL(positionUpdated(QGeoPositionInfo)),this,
     SLOT(nouvellePosition(QGeoPositionInfo)));
    }
}
```

Ce *slot* mettra donc à jour la latitude et la longitude, mais également la date de la mesure, et arrêtera la mesure de la position :

```
void FournisseurGPS::nouvellePosition(const QGeoPositionInfo
     &info)
{
    latitude = info.coordinate().latitude();
    longitude = info.coordinate().longitude();
    ok=true;
    tempsMesure= QDateTime::currentDateTime();
    source->stopUpdates();
}
```

Les opérations issues de l'interface se contenteront de renvoyer les attributs, en vérifiant auparavant s'il ne faut pas prendre une mesure de la position :

```
float FournisseurGPS::Latitude() const {
    if(!mesureOk()) mesure();
    return latitude;
}

float FournisseurGPS::Longitude() const {
    if(!mesureOk()) mesure();
    return longitude;
}
```

Une mesure est considérée comme correcte si elle a moins d'une heure :

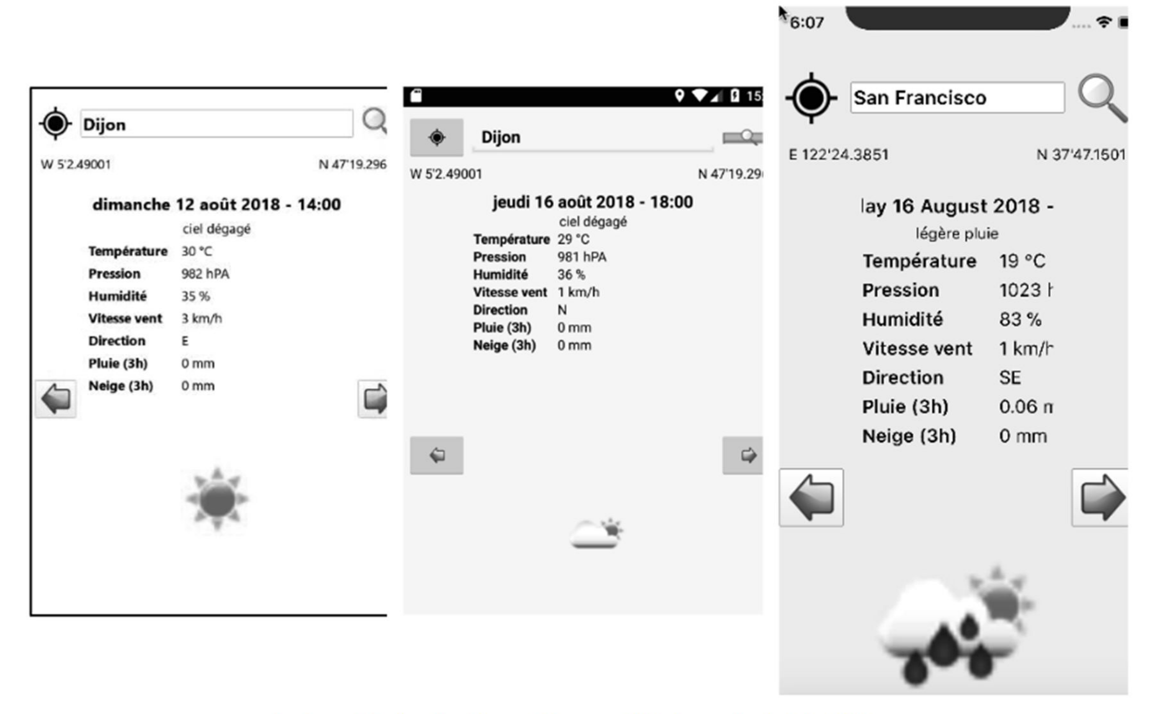
```
bool FournisseurGPS::mesureOk() const {
    QDateTime now = QDateTime::currentDateTime();
    QDateTime before = now.addSecs(-3600);
    return (before < tempsMesure);
}
```

Pour prendre la mesure, il suffit de démarrer la mise à jour de la position, et d'attendre que celle-ci soit correcte :

```cpp
void FournisseurGPS::mesure() const
{
    ok = false;

    if(source)
    {
        source->startUpdates();
        while(!ok)
            QCoreApplication::processEvents();
    }
}
```

Notre application est à présent terminée !



Capture 99 : Application météo sous Windows, Android et iOS

k) Améliorations possibles

Cette application est volontairement très simple et limitée pour pouvoir être décrite totalement en peu de pages… La première version que j'ai déposée sur les différents magasins d'applications est celle décrite dans ces pages. Il est possible que je réalise des mises à jour de l'application, vous

pouvez quant à vous essayer de l'améliorer en ajoutant des fonctionnalités, en modifiant l'IHM, etc…

Voici quelques idées de modifications intéressantes :

- Rendre l'interface plus agréable, plus « responsive »
- Traduire en plusieurs langues (voir page 138)
- Remplacez les boutons pour changer d'horaire par un « glissé de doigt » pour rendre l'application plus ergonomique et moderne
- Permettre d'enregistrer des lieux favoris

XIII.Références

1. **Guidet, Alexandre.** *d'UML à C++.* s.l. : Ellipses, 2013. 2729881875.

2. —. *Programmation objet en langage C++.* s.l. : Ellipses, 2008.

3. **Code Heroes.** Javascript : classes abstraites et interfaces. [En ligne] Code Heroes. https://www.codeheroes.fr/index.php/2017/11/08/js-classes-abstraites-et-interfaces/.

4. **Microsoft.** Tools for Cordova - iOS guide. *Documentation Microsoft.* [En ligne] https://docs.microsoft.com/en-us/visualstudio/cross-platform/tools-for-cordova/first-steps/ios-guide.

5. **Qt.** QMake variables. *Documentation Qt.* [En ligne] http://doc.qt.io/qt-5/qmake-variable-reference.html#winrt-manifest.

6. —. Versions de Qt et plateformes supportées. *Documentation Qt.* [En ligne] https://doc.qt.io/archives/qt-5.10/supported-platforms-and-configurations.html.

7. **Xamarin.** layouts. *Guide développeur Xamarin.Forms.* [En ligne] https://developer.xamarin.com/guides/xamarin-forms/user-interface/controls/layouts/.

8. **Raphael.** CSS3 Grid Layout. [En ligne] AlsaCreations, 2015. https://www.alsacreations.com/article/lire/1388-css3-grid-layout.html.

9. **Android Developers.** Resources. *API Developers.* [En ligne] https://developer.android.com/guide/topics/resources/providing-resources.html.

10. **Cordova.** Plugin cordova globalization. *Documentation Apache Cordova.* [En ligne] https://github.com/apache/cordova-plugin-globalization.

11. **Garvin, Jim.** JQuery Localize. *GitHub.* [En ligne] https://github.com/coderifous/jquery-localize.

12. **ECMA.** ECMAScript Internationalization API. *ECMAScript.* [En ligne] https://www.ecma-international.org/ecma-402/1.0/.

13. **Montemagno, James.** Xam.Plugins.Settings. *NuGet.org.* [En ligne] https://www.nuget.org/packages/Xam.Plugins.Settings/.

14. **Qt.** QSettings. *Qt Documentation.* [En ligne] http://doc.qt.io/qt-5/qsettings.html.

15. **Boggan, Pierce.** Simple cross-platform file IO for iOS, Android and Windows. *Blog Xamarin.com.* [En ligne] 13 04 2016. https://blog.xamarin.com/simple-cross-platform-file-io-for-ios-android-and-windows/.

16. **Xamarin.** Plugins for Xamarin. [En ligne] https://www.xamarin.com/plugins.

17. **Montemagno, James.** Documentation Plugin Xamarin.Geolocator. [En ligne] https://jamesmontemagno.github.io/GeolocatorPlugin/GettingStarted.html.

18. —. Documentation MediaPlugin. *github.* [En ligne] https://github.com/jamesmontemagno/MediaPlugin.

19. **Apache.** Cordova - API Camera. *Documentation Cordova.* [En ligne] http://cordova.apache.org/docs/en/latest/reference/cordova-plugin-camera/index.html.

20. **Qt.** Qt Sensors - compatibility map. *Documentation Qt.* [En ligne] https://doc.qt.io/qt-5/compatmap.html.

21. **West, Matt.** Exploring Javascript Device APIs. *Team Treehouse.* [En ligne] http://blog.teamtreehouse.com/exploring-javascript-device-apis.

22. **Wikipédia.** Hypertext Transfert Protocol. [En ligne] https://fr.wikipedia.org/wiki/Hypertext_Transfer_Protocol.

23. **JQuery.** JQuery. [En ligne] https://jquery.com/.

24. **Boggan, Pierce.** Easy iOS App Provisioning with Fastlane and Visual Studio for Mac. [En ligne] 16 6 2017. https://blog.xamarin.com/easy-ios-app-provisioning-fastlane-visual-studio-mac/.

25. **Microsoft.** Publishing Xamarin.iOS apps to the App Store. [En ligne] 2018. https://docs.microsoft.com/en-us/xamarin/ios/deploy-test/app-

distribution/app-store-distribution/publishing-to-the-app-store?tabs=vsmac.

26. **Pierre.** RVB-CMNJ-TSL conversion-définition. *La photo en faits.* [En ligne] http://www.la-photo-en-faits.com/2013/05/RVB-CMJN-TSL-conversion-definition.html?m=1.

27. **icones.pro.** *Icones pro.* [En ligne] http://icones.pro/.

28. **OpenWeatherMap.** API OpenWeatherMap. [En ligne] 2018. https://openweathermap.org/api.

29. **Gamma, Erich, et al.** *Design Patterns: Elements of Reusable Object-Oriented Software.* s.l. : Addison-Wesley, 1994.

30. **Hébuterne, Perochon.** *Android - guide de développement d'applications Java pour Smartphones et Tablettes.* s.l. : ENI, 2014.

31. **Imbert, Jean-Pierre.** *Développez pour iPhone et iPad.* s.l. : Micro Application, 2013.

32. **Qt.** Qt pour iOS - notes. *Documentation Qt.* [En ligne] http://doc.qt.io/qt-5/platform-notes-ios.html.

33. —. Documentation Qt. *Documentation Qt.* [En ligne] http://doc.qt.io/qt-5/.

34. **Sahiner, Norvan.** Getting Started: iOS Location Tracking and Streaming w/ Swift. [En ligne] https://www.pubnub.com/blog/2015-05-05-getting-started-ios-location-tracking-and-streaming-w-swift-programming-language/.

35. **Qt.** Qt Camera Overview. *Documentation Qt.* [En ligne] http://doc.qt.io/qt-5/cameraoverview.html.

36. **Apache.** Référence plugin Camera. *Documentation Cordova.* [En ligne] https://cordova.apache.org/docs/en/latest/reference/cordova-plugin-camera/.

37. **Xamarin.** Free provisionning. *Xamarin.* [En ligne] https://developer.xamarin.com/guides/ios/getting_started/installation/device_provisioning/free-provisioning/.

38. **Merit Solutions.** Localisation avec Xamarin Forms. *Merit Solutions.* [En ligne] http://www.meritsolutions.com/mobile-development/android-app-localization-using-xamarin-forms-and-resx/.

39. **Apache.org.** Storage. *Cordova.* [En ligne] https://cordova.apache.org/docs/en/latest/cordova/storage/storage.html.

40. **Microsoft.** Publier une application Cordova. *Documentation Microsoft.* [En ligne] https://docs.microsoft.com/en-us/visualstudio/cross-platform/tools-for-cordova/publishing/publish-to-a-store?view=toolsforcordova-2017.

41. **Doudoux, JM.** Sérialisation Java. [En ligne] http://www.jmdoudoux.fr/java/dej/chap-serialisation.htm.

42. **Wikipédia.** Android. *Wikipedia.* [En ligne] https://fr.wikipedia.org/wiki/Android.

43. —. Google Play Store. *Wikipedia.* [En ligne] https://fr.wikipedia.org/wiki/Google_Play.

44. —. Secure Shell. *Wikipedia.* [En ligne] https://fr.wikipedia.org/wiki/Secure_Shell.

45. —. Système de Gestion de Bases de Données. [En ligne] 2018. https://fr.wikipedia.org/wiki/Système_de_gestion_de_base_de_données.

46. —. Fonction anonyme. [En ligne] https://fr.wikipedia.org/wiki/Fonction_anonyme.

47. **Inconnu.** Take Photo Tutorial iOS 8. *ioscreator.* [En ligne] https://www.ioscreator.com/tutorials/take-photo-tutorial-ios8-swift.

48. —. Icones gratuites. [En ligne] http://icones-gratuites.com/.